# CONTENTS

## 1 Introduction to GloW

## 2 Getting Started

2.1 Running Locally ................................................. 5
2.2 Running in the cloud ........................................... 6
2.3 Notebooks embedded in the docs ......................... 6
2.4 Demo notebook .................................................. 6

## 3 Variant Data Manipulation

3.1 Read and Write VCF, Plink, and BGEN with Spark ........ 7
3.2 Read Genome Annotations (GFF3) as a Spark DataFrame .... 10
3.3 Create a Genomics Delta Lake ................................. 12
3.4 Variant Quality Control ........................................ 13
3.5 Sample Quality Control ....................................... 15
3.6 Liftover ............................................................ 18
3.7 Variant Normalization ......................................... 19
3.8 Split Multiallelic Variants .................................... 22
3.9 Merging Variant Datasets ...................................... 24
3.10 Utility Functions ............................................... 25

## 4 Tertiary Analysis

4.1 Parallelizing Command-Line Bioinformatics Tools With the Pipe Transformer 29
4.2 Using Python Statistics Libraries ............................. 32
4.3 Genome-wide Association Study Regression Tests ........... 32
4.4 GloWGR: Whole Genome Regression .......................... 36

## 5 Troubleshooting

## 6 Blog Posts

6.1 Introducing GloWGR: An industrial-scale, ultra-fast and sensitive method for genetic association studies 47
6.2 GloW 0.4 Enables Integration of Genomic Variant and Annotation Data .............................. 51
6.3 GloW 0.3.0 Introduces Several New Large-Scale Genomic Analysis Features .......................... 58
6.4 Streamlining Variant Normalization on Large Genomic Datasets ........................................ 63

## 7 Additional Resources

7.1 Databricks notebooks ......................................... 69
7.2 External blog posts .............................................. 69

## 8 Python API

8.1 Glow Top-Level Functions ..................................... 71
8.2 Glow PySpark Functions ....................................... 72
Glow is an open-source toolkit for working with genomic data at biobank-scale and beyond. The toolkit is natively built on Apache Spark, the leading unified engine for big data processing and machine learning, enabling genomics workflows to scale to population levels.
Genomics data has been doubling every seven months globally. It has now reached a scale where genomics has become a big data problem. However, most of the tools for working with genomics data are built to work on single nodes and will not scale. Furthermore, it has become challenging for scientists to manage storage of public data.

Glow solves these problems by bridging bioinformatics and the big data ecosystem. It enables bioinformaticians and computational biologists to leverage best practices used by data engineers and data scientists across industry.

Under the hood, Glow is built on Apache Spark and Delta Lake, enabling distributed computation on and distributed storage of genotype data. The library is backwards compatible with genomics file formats and bioinformatics tools developed in academia, enabling users to easily share data with collaborators.

Glow is used to:

- Ingest public genotype data into a data lake that acts as a single source of truth.
- Run quality control, statistical analysis, and association studies on population-scale datasets.
- Build reproducible, production-grade genomics data pipelines that will scale to petabytes and beyond.

Glow features:

- Genomic datasources: To read datasets in common file formats such as VCF, BGEN, and Plink into Spark DataFrames.
- Genomic functions: Common operations such as computing quality control statistics, running regression tests, and performing simple transformations are provided as Spark functions that can be called from Python, SQL, Scala, or R.
• Data preparation building blocks: Glow includes transformations such as variant normalization and lift over to help produce analysis ready datasets.

• Integration with existing tools: With Spark, you can write user-defined functions (UDFs) in Python, R, SQL, or Scala. Glow also makes it easy to run DataFrames through command line tools.

• Integration with other data types: Genomic data can generate additional insights when joined with data sets such as electronic health records, real world evidence, and medical images. Since Glow returns native Spark SQL DataFrames, its simple to join multiple data sets together.
CHAPTER
TWO

GETTING STARTED

2.1 Running Locally

Glow requires Apache Spark 2.4.3 (or a later version of Spark 2.4.x that is built on Scala 2.11).

Python

If you don’t have a local Apache Spark installation, you can install it from PyPI:

```
$ pip install pyspark==2.4.3
```

or download a specific distribution.

Install the Python frontend from pip:

```
$ pip install glow.py
```

and then start the Spark shell with the Glow maven package:

```
$ ./bin/pyspark --packages io.projectglow:glow_2.11:0.5.0
--conf spark.hadoop.io.compression.codecs=io.projectglow.sql.util.BGZFCCodec
```

To start a Jupyter notebook instead of a shell:

```
$ PYSPARK_DRIVER_PYTHON=jupyter PYSPARK_DRIVER_PYTHON_OPTS=notebook ./bin/pyspark --packages io.projectglow:glow_2.11:0.5.0
--conf spark.hadoop.io.compression.codecs=io.projectglow.sql.util.BGZFCCodec
```

And now your notebook is glowing! To access the Glow functions, you need to register them with the Spark session.

```
import glow
glow.register(spark)
df = spark.read.format('vcf').load(path)
```

Scala

If you don’t have a local Apache Spark installation, download a specific distribution.

Start the Spark shell with the Glow maven package:

```
$ ./bin/spark-shell --packages io.projectglow:glow_2.11:0.5.0
--conf spark.hadoop.io.compression.codecs=io.projectglow.sql.util.BGZFCCodec
```

To access the Glow functions, you need to register them with the Spark session.
import io.projectglow.Glow
Glow.register(spark)
val df = spark.read.format("vcf").load(path)

2.2 Running in the cloud

The easiest way to use Glow in the cloud is with the Databricks Runtime for Genomics. However, it works with any cloud provider or Spark distribution. You need to install the maven package io.project:glow_2.11:$\{version}\$ and optionally the Python frontend glow.py. Also set the Spark configuration spark.hadoop.io.compression.codecs to io.projectglow.sql.util.BGZFCCodec in order to read and write BGZF-compressed files.

2.3 Notebooks embedded in the docs

To demonstrate example use cases of Glow functionalities, most doc pages are accompanied by embedded Databricks Notebooks. Most of the code in these notebooks can be run on Spark and Glow alone, but a few functions such as display() or dbutils() are only available on Databricks. See Databricks notebooks for more info.

Also note that the path to datasets used as example in these notebooks is usually a folder in /databricks-datasets/genomics/ and should be replaced with the appropriate path based on your own folder structure.

2.4 Demo notebook

This notebook showcases some of the key functionality of Glow, like reading in a genomic dataset, saving it as a Delta Lake, and performing a genome-wide association study.

2.4.1 Notebook

<a href="../additional-resources.html#running-databricks-notebooks">How to run a notebook</a> <a href="./_static/notebooks/tertiary/gwas.html">Get notebook link</a>
VARIANT DATA MANIPULATION

Glow offers functionalities to extract, transform and load (ETL) genomic variant data into Spark DataFrames, enabling seamless manipulation, filtering, quality control and transformation between file formats.

3.1 Read and Write VCF, Plink, and BGEN with Spark

Glow makes it possible to read and write variant data at scale using Spark SQL.

Tip: This topic uses the terms “variant” or “variant data” to refer to single nucleotide variants and short indels.

3.1.1 VCF

You can use Spark to read VCF files just like any other file format that Spark supports through the DataFrame API using Python, R, Scala, or SQL.

```python
df = spark.read.format("vcf").load(path)
```

The returned DataFrame has a schema that mirrors a single row of a VCF. Information that applies to an entire variant (SNV or indel), such as the contig name, start and end positions, and INFO attributes, is contained in columns of the DataFrame. The genotypes, which correspond to the GT FORMAT fields in a VCF, are contained in an array with one entry per sample. Each entry is a struct with fields that are described in the VCF header.

The path that you provide can be the location of a single file, a directory that contains VCF files, or a Hadoop glob pattern that identifies a group of files. Sample IDs are not included by default. See the parameters table below for instructions on how to include them.

You can control the behavior of the VCF reader with a few parameters. All parameters are case insensitive.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>includeSampleIds</td>
<td>boolean</td>
<td>true</td>
<td>If true, each genotype includes the name of the sample ID it belongs to. Sample names increase the size of each row, both in memory and on storage.</td>
</tr>
<tr>
<td>flattenInfoFields</td>
<td>boolean</td>
<td>true</td>
<td>If true, each info field in the input VCF will be converted into a column in the output DataFrame with each column typed as specified in the VCF header. If false, all info fields will be contained in a single column with a string -&gt; string map of info keys to values.</td>
</tr>
<tr>
<td>validationStringency</td>
<td>string</td>
<td>silent</td>
<td>Controls the behavior when parsing a malformed row. If silent, the row will be dropped silently. If lenient, the row will be dropped and a warning will be logged. If strict, an exception will be thrown and reading will fail.</td>
</tr>
</tbody>
</table>
**Note:** Starting from Glow 0.4.0, the `splitToBiallelic` option for the VCF reader no longer exists. To split multiallelic variants to biallelics use the `split_multiallelic` transformer after loading the VCF.

**Important:** The VCF reader uses the 0-start, half-open (zero-based) coordinate system.

You can save a DataFrame as a VCF file, which you can then read with other tools. To write a DataFrame as a single VCF file specify the format "bigvcf":

```python
df.write.format("bigvcf").save(path)
```

The file extension of the output path determines which, if any, compression codec should be used. For instance, writing to a path such as `/genomics/my_vcf.vcf.bgz` will cause the output file to be block gzipped.

If you’d rather save a sharded VCF where each partition saves to a separate file:

```python
df.write.format("vcf").save(path)
```

To control the behavior of the sharded VCF writer, you can provide the following option:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>compression</td>
<td>string</td>
<td>n/a</td>
<td>A compression codec to use for the output VCF file. The value can be the full name of a compression codec class (for example <code>GzipCodec</code>) or a short alias (for example <code>gzip</code>). To use the block gzip codec, specify <code>bgzf</code>.</td>
</tr>
</tbody>
</table>

For both the single and sharded VCF writer, you can use the following options:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vcfHeader</td>
<td>string</td>
<td>infer</td>
<td>If <code>infer</code>, infers the header from the DataFrame schema. This value can be a complete header starting with <code>##</code> or a Hadoop filesystem path to a VCF file. The header from this file is used as the VCF header for each partition.</td>
</tr>
<tr>
<td>validationStringency</td>
<td>string</td>
<td>silent</td>
<td>Controls the behavior when parsing a malformed row. If <code>silent</code>, the row will be dropped silently. If <code>lenient</code>, the row will be dropped and a warning will be logged. If <code>strict</code>, an exception will be thrown and writing will fail.</td>
</tr>
</tbody>
</table>

If the header is inferred from the DataFrame, the sample IDs are derived from the rows. If the sample IDs are missing, they will be represented as `sample_n`, for which `n` reflects the index of the sample in a row. In this case, there must be the same number of samples in each row.

- For the big VCF writer, the inferred sample IDs are the distinct set of all sample IDs from the DataFrame.
- For the sharded VCF writer, the sample IDs are inferred from the first row of each partition and must be the same for each row. If the rows do not contain the same samples, provide a complete header of a filesystem path to a VCF file.

### 3.1.2 BGEN

Glow provides the ability to read BGEN files, including those distributed by the UK Biobank project.
df = spark.read.format("bgen").load(path)

As with the VCF reader, the provided path can be a file, directory, or glob pattern. If .bgi index files are located in the same directory as the data files, the reader uses the indexes to more efficiently traverse the data files. Data files can be processed even if indexes do not exist. The schema of the resulting DataFrame matches that of the VCF reader.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>useBgenIndex</td>
<td>boolean</td>
<td>true</td>
<td>If true, use .bgi index files.</td>
</tr>
<tr>
<td>sampleFilePath</td>
<td>string</td>
<td>n/a</td>
<td>Path to a .sample Oxford sample information file containing sample IDs if not stored in the BGEN file.</td>
</tr>
<tr>
<td>sampleIdColumn</td>
<td>string</td>
<td>ID_2</td>
<td>Name of the column in the .sample file corresponding to the sample IDs.</td>
</tr>
</tbody>
</table>

You can use the DataFrameWriter API to save a single BGEN file, which you can then read with other tools.

df.write.format("bigbgen").save(path)

If the genotype arrays are missing ploidy and/or phasing information, the BGEN writer infers the values using the provided values for ploidy, phasing, or posteriorProbabilities in the genotype arrays. You can provide the value for ploidy using an integer value ploidy or it can be inferred using the length of an array calls, and you can provide the phasing information using a boolean value phased.

To control the behavior of the BGEN writer, you can provide the following options:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitsPerProbability</td>
<td>integer</td>
<td>16</td>
<td>Number of bits used to represent each probability value. Must be 8, 16, or 32.</td>
</tr>
<tr>
<td>maximumInferredPloidy</td>
<td>integer</td>
<td>10</td>
<td>The maximum ploidy that will be inferred for unphased data if ploidy is missing.</td>
</tr>
<tr>
<td>defaultInferredPloidy</td>
<td>integer</td>
<td>2</td>
<td>The inferred ploidy if phasing and ploidy are missing, or ploidy is missing and cannot be inferred from posteriorProbabilities.</td>
</tr>
<tr>
<td>defaultInferredPhasing</td>
<td>boolean</td>
<td>false</td>
<td>The inferred phasing if phasing is missing and cannot be inferred from posteriorProbabilities.</td>
</tr>
</tbody>
</table>

### 3.1.3 PLINK

Glow provides the ability to read binary PLINK binary PED (BED) files with accompanying BIM and FAM files. The provided path can be a file or glob pattern.

df = spark.read.format("plink").load("{prefix}.bed".format(prefix=prefix))

The schema of the resulting DataFrame matches that of the VCF reader. The accompanying variant and sample information files must be located at {prefix}.bim and {prefix}.fam.
### 3.2 Read Genome Annotations (GFF3) as a Spark DataFrame

GFF3 (Generic Feature Format Version 3) is a 9-column tab-separated text file format commonly used to store genomic annotations. Typically, the majority of annotation data in this format appears in the ninth column, called attributes, as a semi-colon-separated list of \texttt{tag} = \texttt{value} entries. If Spark’s standard \texttt{csv} data source is used to read GFF3 files, the whole list of attribute tag-value pairs will be read as a single string-typed column, making queries on these tags/values cumbersome.

To address this issue, Glow provides the \texttt{gff} data source. In addition to loading the first 8 columns of GFF3 as properly typed columns, the \texttt{gff} data source is able to parse all attribute tag-value pairs in the ninth column of GFF3 and create an appropriately typed column for each tag. In each row, the column corresponding to a tag will contain the tag’s value in that row (or \texttt{null} if the tag does not appear in the row).

Like any Spark data source, reading GFF3 files using the \texttt{gff} data source can be done in a single line of code:

```python
df = spark.read.format("gff").load(path)
```

The \texttt{gff} data source supports all compression formats supported by Spark’s \texttt{csv} data source, including .gz and .bgz files. It also supports reading globs of files in one command.

**Note:** The \texttt{gff} data source ignores any comment and directive lines (lines starting with \#) in the GFF3 file as well as any FASTA lines that may appear at the end of the file.

### 3.2.1 Schema

#### 1. Inferred schema

If no user-specified schema is provided (as in the example above), the data source infers the schema as follows:

- The first 8 fields of the schema (“base” fields) correspond to the first 8 columns of the GFF3 file. Their names, types and order will be as shown below:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>includeSampleIds</td>
<td>boolean</td>
<td>true</td>
<td>If true, each genotype includes the name of the sample ID it belongs to.</td>
</tr>
<tr>
<td>bimDelimiter</td>
<td>string</td>
<td>(tab)</td>
<td>Whitespace delimiter in the \texttt{(prefix).bim} file.</td>
</tr>
<tr>
<td>famDelimiter</td>
<td>string</td>
<td>(space)</td>
<td>Whitespace delimiter in the \texttt{(prefix).fam} file.</td>
</tr>
<tr>
<td>mergeFidIid</td>
<td>boolean</td>
<td>true</td>
<td>If true, sets the sample ID to the family ID and individual ID merged with an underscore delimiter. If false, sets the sample ID to the individual ID.</td>
</tr>
<tr>
<td>seqId: string (nullable = true)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>source: string (nullable = true)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>type: string (nullable = true)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>start: long (nullable = true)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>end: long (nullable = true)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>score: double (nullable = true)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>strand: string (nullable = true)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>phase: integer (nullable = true)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: Although the start column in the GFF3 file is 1-based, the start field in the DataFrame will be 0-based to match the general practice in Glow.

- The next fields in the inferred schema will be created as the result of parsing the attributes column of the GFF3 file. Each tag will have its own field in the schema. Fields corresponding to any “official” tag (those referred to as tags with pre-defined meaning) come first, followed by fields corresponding to any other tag (“unofficial” tags) in alphabetical order.

The complete list of official fields, their data types, and order are as shown below:

| ID: string (nullable = true) |
| Name: string (nullable = true) |
| Alias: string (nullable = true) |
| Parent: array (nullable = true) |
| element: string (containsNull = true) |
| Target: string (nullable = true) |
| Gap: string (nullable = true) |
| DerivesFrom: string (nullable = true) |
| Note: array (nullable = true) |
| element: string (containsNull = true) |
| Dbxref: array (nullable = true) |
| element: string (containsNull = true) |
| OntologyTerm: array (nullable = true) |
| element: string (containsNull = true) |
| Is_circular: boolean (nullable = true) |

The unofficial fields will be of string type.

Note:
- If any of official tags does not appear in any row of the GFF3 file, the corresponding field will be excluded from the inferred schema.
- The official/unofficial field name will be exactly as the corresponding tag appears in the GFF3 file (in terms of letter case).
- The parser is insensitive to the letter case of the tag, e.g., if the attributes column in the GFF3 file contains both note and Note tags, they will be both mapped to the same column in the DataFrame. The name of the column in this case will be either note or Note, chosen randomly.

2. User-specified schema

As with any Spark data source, the gff data source is also able to accept a user-specified schema through the .schema command. The user-specified schema can have any subset of the base, official, and unofficial fields. The data source is able to read only the specified base fields and parse out only the specified official and unofficial fields.
from the attributes column of the GFF3 file. Here is an example of how the user can specify some base, official, and unofficial fields while reading the GFF3 file:

```python
mySchema = StructType(
    [StructField('seqId', StringType()),          # Base field
     StructField('start', LongType()),           # Base field
     StructField('end', LongType()),             # Base field
     StructField('ID', StringType()),            # Official field
     StructField('Dbxref', ArrayType(StringType())),  # Official field
     StructField('mol_type', StringType())       # Unofficial field
    ]
)

df_user_specified = spark.read.format("gff").schema(mySchema).load(path)
```

Note:

- The base field names in the user-specified schema must match the names in the list above in a case-sensitive manner.
- The official and unofficial fields will be matched with their corresponding tags in the GFF3 file in a case-and-underscore-insensitive manner. For example, if the GFF3 file contains the official tag `db_xref`, a user-specified schema field with the name `dbxref`, `Db_Xref`, or any other case-and-underscore-insensitive match will correspond to that tag.
- The user can also include the original attributes column of the GFF3 file as a string field by including `StructField('attributes', StringType())` in the schema.

### 3.3 Create a Genomics Delta Lake

Genomics data is usually stored in specialized flat-file formats such as VCF or BGEN.

The example below shows how to ingest a VCF into a genomics Delta Lake table using Glow in Python (R, Scala, and SQL are also supported).

You can use Delta tables for second-latency queries, performant range-joins (similar to the single-node bioinformatics tool bedtools intersect), aggregate analyses such as calculating summary statistics, machine learning or deep learning.

Tip: We recommend ingesting VCF files into Delta tables once volumes reach >1000 samples, >10 billion genotypes or >1 terabyte.
3.3.1 VCF to Delta Lake table notebook

How to run a notebook

Get notebook link

3.4 Variant Quality Control

Glow includes a variety of tools for variant quality control.

Tip: This topic uses the terms “variant” or “variant data” to refer to single nucleotide variants and short indels.

You can calculate quality control statistics on your variant data using Spark SQL functions, which can be expressed in Python, R, Scala, or SQL.
### Function | Arguments | Return
---|---|---
hardey-weinberg | The `genotypes` array. This function assumes that the variant has been converted to a biallelic representation. | A struct with two elements: the expected heterozygous frequency according to Hardy-Weinberg equilibrium and the associated p-value. |
call_summary_stats | The `genotypes` array | A struct containing the following summary stats:
- `callRate`: The fraction of samples with a called genotype
- `nCalled`: The number of samples with a called genotype
- `nUncalled`: The number of samples with a missing or un-called genotype, as represented by a `.` in a VCF or `-1` in a DataFrame.
- `nHet`: The number of heterozygous samples
- `nHomzygous`: An array with the number of samples that are homozygous for each allele. The 0th element describes how many sample are hom-ref.
- `nNonRef`: The number of samples that are not hom-ref
- `nAllelesCalled`: An array with the number of times each allele was seen
- `alleleFrequencies`: An array with the frequency for each allele

dp_summary_stats | The `genotypes` array | A struct containing the min, max, mean, and sample standard deviation for genotype depth (DP in VCF v4.2 specification) across all samples |
gq_summary_stats | The `genotypes` array | A struct containing the min, max, mean, and sample standard deviation for genotype quality (GQ in VCF v4.2 specification) across all samples |

#### 3.4.1 Notebook

<a href="../additional-resources.html#running-databricks-notebooks">How to run a notebook</a> <a href="../_static/notebooks/etl/variant-qc-demo.html">Get notebook link</a> <a href="../_static/notebooks/etl/variant-qc-demo.html" class="loading-spinner" id='1417486713135321750' height='1000px' width='100%'"/>
3.5 Sample Quality Control

You can calculate quality control statistics on your variant data using Spark SQL functions, which can be expressed in Python, R, Scala, or SQL.

Each of these functions returns an array of structs containing metrics for one sample. If sample ids are including in the input DataFrame, they will be propagated to the output. The functions assume that the genotypes in each row of the input DataFrame contain the same samples in the same order.
### Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Arguments</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>sample_call_summary_stats</td>
<td>referenceAllele string, alternateAlleles array of strings, genotypes array calls</td>
<td>A struct containing the following summary stats:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• callRate: The fraction of variants where this sample has a called genotype. Equivalent to nCalled / (nCalled + nUncalled)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• nCalled: The number of variants where this sample has a called genotype</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• nUncalled: The number of variants where this sample does not have a called genotype</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• nHomRef: The number of variants where this sample is homozygous reference</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• nHet: The number of variants where this sample is heterozygous</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• nHomVar: The number of variants where this sample is homozygous non reference</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• nSnv: The number of calls where this sample has a single nucleotide variant. This value is the sum of nTransition and nTransversion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• nInsertion: Insertion variant count</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• nDeletion: Deletion variant count</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• nTransition: Transition count</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• nTransversion: Transversion count</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• nSpanningDeletion: The number of calls where this sample has a spanning deletion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• rTiTv: Ratio of transitions to transversions (nTransition / nTransversion)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• rInsertionDeletion: Ratio of insertions to deletions (nInsertion / nDeletion)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• rHetHomVar: Ratio of heterozygous to homozygous variant calls (nHet / nHomVar)</td>
</tr>
<tr>
<td>sample_dp_summary_stats</td>
<td>genotypes array with a depth field</td>
<td>A struct with min, max, mean, and stddev</td>
</tr>
<tr>
<td>sample_gq_summary_stats</td>
<td>genotypes array with a conditionalQuality field</td>
<td>A struct with min, max, mean, and stddev</td>
</tr>
</tbody>
</table>
3.5.1 Computing user-defined sample QC metrics

In addition to the built-in QC functions discussed above, Glow provides two ways to compute user-defined per-sample statistics.

**aggregate_by_index**

First, you can aggregate over each sample in a genotypes array using the `aggregate_by_index` function.

```
aggregate_by_index(array, initial_state, update_function, merge_function, eval_function)
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>array&lt;T&gt;</td>
<td>An array-typed column. There are no requirements on the element datatype. This array is expected to be the same length for each row in the input DataFrame. The output of <code>aggregate_by_index</code> is an array with the same length as each input row.</td>
</tr>
<tr>
<td>initial_state</td>
<td>U</td>
<td>The initial aggregation state for each sample.</td>
</tr>
<tr>
<td>update_function</td>
<td>T&gt; U</td>
<td>A function that returns a new single sample aggregation state given the current aggregation state and a new data element.</td>
</tr>
<tr>
<td>merge_function</td>
<td>U&gt; U</td>
<td>A function that combines two single sample aggregation states. This function is necessary since the aggregation is computed in a distributed manner across all nodes in the cluster.</td>
</tr>
<tr>
<td>eval_function (optional)</td>
<td>U -&gt; V</td>
<td>A function that returns the output for a sample given that sample’s aggregation state. This function is optional. If it is not specified, the aggregation state will be returned.</td>
</tr>
</tbody>
</table>

For example, this code snippet uses `aggregate_by_index` to compute the mean for each array position:

```
aggregate_by_index(
    array_col,
    (0d, 0l),
    (state, element) -> (state.col1 + element, state.col2 + 1),
    (state1, state2) -> (state1.col1 + state2.col1, state1.col2 + state2.col2),
    state -> state.col1 / state.col2)
```

**Explode and aggregate**

If your dataset is not in a normalized, pVCF-esque shape, or if you want the aggregation output in a table rather than a single array, you can explode the genotypes array and use any of the aggregation functions built into Spark. For example, this code snippet computes the number of sites with a non-reference allele for each sample:

```
import pyspark.sql.functions as fx
exploded_df = df.withColumn("genotype", fx.explode("genotypes"))
    .withColumn("hasNonRef", fx.expr("exists(genotype.calls, call -> call != -1 and call != 0)"))
agg = exploded_df.groupBy("genotype.sampleId", "hasNonRef")
    .agg(fx.count(fx.lit(1)))
    .orderBy("sampleId", "hasNonRef")
```
3.6 Liftover

Liftover converts genomic data between reference assemblies. The UCSC liftover tool uses a chain file to perform simple coordinate conversion, for example on BED files. The Picard LiftOverVcf tool also uses the new reference assembly file to transform variant information (e.g. alleles and INFO fields). Glow can be used to run coordinate liftover and variant liftover.

3.6.1 Create a liftover cluster

For both coordinate and variant liftover, you need a chain file on every node of the cluster. On a Databricks cluster, an example of a cluster-scoped init script you can use to download the required file for liftover from the b37 to the hg38 reference assembly is as follows:

```bash
#!/usr/bin/env bash
set -ex
set -o pipefail
mkdir /opt/liftover
curl https://raw.githubusercontent.com/broadinstitute/gatk/master/scripts/funcotator/data_sources/gnomAD/b37ToHg38.over.chain --output /opt/liftover/b37ToHg38.over.chain
```

3.6.2 Coordinate liftover

To perform liftover for genomic coordinates, use the function `lift_over_coordinates`. `lift_over_coordinates`, which has the following parameters.

- `chromosome`: string
- `start`: long
- `end`: long
- `chain file`: string (constant value, such as one created with `lit()`)
- `minimum fraction of bases that must remap`: double (optional, defaults to .95)

The returned `struct` has the following values if liftover succeeded. If not, the function returns `null`.

- `contigName`: string
- `start`: long
- `end`: long

```scala
output_df = input_df.withColumn('lifted', glow.lift_over_coordinates('contigName', 'start', 'end', chain_file, 0.99))
```
3.6.3 Variant liftOver

For genetic variant data, use the `lift_over_variants` transformer. In addition to performing liftOver for genetic coordinates, variant liftOver performs the following transformations:

- Reverse-complement and left-align the variant if needed
- Adjust the SNP, and correct allele-frequency-like INFO fields and the relevant genotypes if the reference and alternate alleles have been swapped in the new genome build

Pull a target assembly reference file down to every node in the Spark cluster in addition to a chain file before performing variant liftOver.

The `lift_over_variants` transformer operates on a DataFrame containing genetic variants and supports the following options:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chain_file</td>
<td>n/a</td>
<td>The path of the chain file.</td>
</tr>
<tr>
<td>reference_file</td>
<td>n/a</td>
<td>The path of the target reference file.</td>
</tr>
<tr>
<td>min_match_ratio</td>
<td>.95</td>
<td>Minimum fraction of bases that must remap.</td>
</tr>
</tbody>
</table>

The output DataFrame’s schema consists of the input DataFrame’s schema with the following fields appended:

- INFO_SwappedAlleles: boolean (null if liftOver failed, true if the reference and alternate alleles were swapped, false otherwise)
- INFO_ReverseComplementedAlleles: boolean (null if liftover failed, true if the reference and alternate alleles were reverse complemented, false otherwise)
- liftOverStatus: struct
  - success: boolean (true if liftOver succeeded, false otherwise)
  - errorMessage: string (null if liftOver succeeded, message describing reason for liftOver failure otherwise)

If liftOver succeeds, the output row contains the liftOver result and `liftOverStatus.success` is true. If liftOver fails, the output row contains the original input row, the additional INFO fields are null, `liftOverStatus.success` is false, and `liftOverStatus.errorMessage` contains the reason liftOver failed.

```python
output_df = glow.transform('lift_over_variants', input_df, chain_file=chain_file,
reference_file=reference_file)
```

Liftover notebook

<iframe src="../_static/notebooks/etl/lift-over.html" id="102679705915719020" height="1000px" width="100%" style="overflow-y:hidden;" scrolling="no"></iframe>

3.7 Variant Normalization

Different genomic analysis tools often represent the same genomic variant in different ways, making it non-trivial to integrate and compare variants across call sets. Therefore, **variant normalization** is an essential step to be applied on variants before further downstream analysis to make sure the same variant is represented identically in different call...
sets. Normalization is achieved by making sure the variant is parsimonious and left-aligned (see Variant Normalization for more details).

Glow provides variant normalization capabilities as a DataFrame transformer as well as a SQL expression function with a Python API, bringing unprecedented scalability to this operation.

**Note:** Glow’s variant normalization algorithm follows the same logic as those used in normalization tools such as bcftools norm and vt normalize. This normalization logic is different from the one used by GATK’s LeftAlignAndTrimVariants, which sometimes yields incorrect normalization (see Variant Normalization for more details).

### 3.7.1 normalize_variants Transformer

The `normalize_variants` transformer can be applied to normalize a variant DataFrame, such as one generated by loading VCF or BGEN files. The output of the transformer is described under the `replace_columns` option below.

### 3.7.2 Usage

Assuming `df_original` is a variable of type DataFrame which contains the genomic variant records, and `ref_genome_path` is a variable of type String containing the path to the reference genome file, a minimal example of using this transformer for normalization is as follows:

**Python**

```python
df_normalized = glow.transform("normalize_variants", df_original, reference_genome_path=ref_genome_path)
```

**Scala**

```scala
df_normalized = Glow.transform("normalize_variants", df_original, Map("reference_genome_path" -> ref_genome_path))
```

### 3.7.3 Options

The `normalize_variants` transformer has the following options:
<table>
<thead>
<tr>
<th>Option</th>
<th>Type</th>
<th>Possible values and description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference_genome_path</td>
<td>string</td>
<td>Path to the reference genome .fasta or .fa file. This file must be accompanied with a .fai index file in the same folder.</td>
</tr>
</tbody>
</table>
| replace_columns     | boolean  | False: The transformer does not modify the original start, end, referenceAllele and alternateAlleles columns. Instead, a StructType column called normalizationResult is added to the DataFrame. This column contains the normalized start, end, referenceAllele, and alternateAlleles columns as well as the normalizationStatus StructType as the fifth field, which contains the following subfields:  
  - changed: Indicates whether the variant data was changed as a result of normalization  
  - errorMessage: An error message in case the attempt at normalizing the row hit an error. In this case, the changed field will be set to False. If no errors occur this field will be null. In case of error, the first four fields in normalizationResult will be null.  
  
  True (default): The original start, end, referenceAllele, and alternateAlleles columns are replaced with the normalized values in case they have changed. Otherwise (in case of no change or an error), the original start, end, referenceAllele, and alternateAlleles are not modified. A StructType normalizationStatus column is added to the DataFrame with the same subfields explained above. |
| mode (deprecated)   | string   | normalize: Only normalizes the variants (if user does not pass the option, normalize is assumed as default) |

3.7. Variant Normalization
3.7.4 normalize_variant Function

The normalizer can also be used as a SQL expression function. See Glow PySpark Functions for details on how to use it in the Python API. An example of an expression using the normalize_variant function is as follows:

```python
from pyspark.sql.functions import expr

normalization_expr = "normalize_variant(contigName, start, end, referenceAllele, alternateAlleles, '{ref_genome}')".format(ref_genome=ref_genome_path)

df_normalized = df_original.withColumn('normalizationResult', expr(normalization_expr))
```

3.8 Split Multiallelic Variants

Splitting multiallelic variants to biallelic variants is a transformation sometimes required before further downstream analysis. Glow provides the split_multiallelics transformer to be applied on a variant DataFrame to split multiallelic variants in the DataFrame to biallelic variants. This transformer is able to handle any number of ALT alleles and any ploidy.

**Note:** The splitting logic used by the split_multiallelics transformer is the same as the one used by the vt decompose tool of the vt package with option `-s` (note that the example provided at vt decompose user manual page does not reflect the behavior of `vt decompose -s` completely correctly).

The precise behavior of the split_multiallelics transformer is presented below:

- A given multiallelic row with $n$ ALT alleles is split to $n$ biallelic rows, each with one of the ALT alleles of the original multiallelic row. The REF allele in all split rows is the same as the REF allele in the multiallelic row.

- Each INFO field is appropriately split among split rows if it has the same number of elements as number of ALT alleles, otherwise it is repeated in all split rows. The boolean INFO field splitFromMultiAllelic is added/modified to reflect whether the new row is the result of splitting a multiallelic row through this transformation or not. A new INFO field called OLD_MULTIALLELIC is added to the DataFrame, which for each split row, holds the CHROM:POS:REF/ALT of its original multiallelic row. Note that the INFO field must be flattened (as explained here) in order to be split by this transformer. Unflattened INFO fields (such as those inside an attributes field) will not be split, but just repeated in whole across all split rows.

- Genotype fields for each sample are treated as follows: The GT field becomes biallelic in each row, where the original ALT alleles that are not present in that row are replaced with no call. The fields with number of entries equal to number of REF + ALT alleles, are properly split into rows, where in each split row, only entries corresponding to the REF allele as well as the ALT allele present in that row are kept. The fields which follow colex order (e.g., GL, PL, and GP) are properly split between split rows where in each row only the elements corresponding to genotypes comprising of the REF and ALT alleles in that row are listed. Other genotype fields are just repeated over the split rows.

- Any other field in the DataFrame is just repeated across the split rows.

As an example (shown in VCF file format), the following multiallelic row
will be split into the following two biallelic rows:

```
#CHROM POS ID REF ALT QUAL FILTER INFO FORMAT SAMPLE1
20 101 . A ACCA . PASS VC=INDEL;AC=3,2;AF=0.375,0.25;AN=8 GT:AD:DP:GQ:PL 0/1:2,15:31:30:99:2407,0,533,697,822,574
20 101 . A TCGG . PASS VC=INDEL;AC=2;AF=0.25;AN=8;
```

### 3.8.1 Usage

Assuming `df_original` is a variable of type DataFrame which contains the genomic variant records, an example of using this transformer for splitting multiallelic variants is:

**Python**

```python
df_split = glow.transform("split_multiallelics", df_original)
```

**Scala**

```scala
df_split = Glow.transform("split_multiallelics", df_original)
```

**Tip:** The `split_multiallelics` transformer is often significantly faster if the `whole-stage code generation` feature of Spark Sql is turned off. Therefore, it is recommended that you temporarily turn off this feature using the following command before using this transformer.

**Python**

```python
spark.conf.set("spark.sql.codegen.wholeStage", False)
```

**Scala**

```scala
spark.conf.set("spark.sql.codegen.wholeStage", false)
```

Remember to turn this feature back on after your split DataFrame is materialized.

---

**Split Multiallelic Variants notebook**

```html
<iframe src="../_static/notebooks/etl/splitmultiallelics-transformer.html" id="9025250340857958265" height="1000px" width="100%" style="overflow-y:hidden;" scrolling="no"></iframe>
```
3.9 Merging Variant Datasets

You can use Glow and Spark to merge genomic variant datasets from non-overlapping sample sets into a multi-sample dataset. In these examples, we will read from VCF files, but the same logic works on *DataFrames backed by other file formats*.

First, read the VCF files into a single Spark DataFrame:

```python
from pyspark.sql.functions import *

df = spark.read.format('vcf').load([path1, path2])

# Alternatively, you can use the "union" DataFrame method if the VCF files have the same schema
df1 = spark.read.format('vcf').load(path1)
df2 = spark.read.format('vcf').load(path2)
df = df1.union(df2)
```

The resulting DataFrame contains all records from the VCFs you want to merge, but the genotypes from different samples at the same site have not been combined. You can use an aggregation to combine the genotype arrays.

```python
from pyspark.sql.functions import *

merged_df = df.groupBy('contigName', 'start', 'end', 'referenceAllele', 'alternateAlleles')
                .agg(sort_array(flatten(collect_list('genotypes'))).alias('genotypes'),
                     sum('INFO_DP').alias('INFO_DP'))
```

**Important:** When reading VCF files for a merge operation, *sampleId* must be the first field in the genotype struct. This is the default Glow schema.

The genotypes from different samples now appear in the same genotypes array.

**Note:** If the VCFs you are merging contain different sites, elements will be missing from the genotypes array after aggregation. Glow automatically fills in missing genotypes when writing to *bigvcf*, so an exported VCF will still contain all samples.

### 3.9.1 Aggregating INFO fields

To preserve INFO fields in a merge, you can use the aggregation functions in Spark. For instance, to emit an *INFO_DP* column that is the sum of the *INFO_DP* columns across all samples:

```python
from pyspark.sql.functions import *

merged_df = df.groupBy('contigName', 'start', 'end', 'referenceAllele', 'alternateAlleles')
                .agg(sort_array(flatten(collect_list('genotypes'))).alias('genotypes'),
                     sum('INFO_DP').alias('INFO_DP'))
```
3.9.2 Joint genotyping

The merge logic in this document allows you to quickly aggregate genotyping array data or single sample VCFs. For a more sophisticated aggregation that unifies alleles at overlapping sites and uses cohort-level statistics to refine genotype calls, we recommend running a joint genotyping pipeline like the one included in the Databricks Runtime for Genomics.

3.10 Utility Functions

Glow includes a variety of utility functions for performing basic data manipulation.

3.10.1 Struct transformations

Glow’s struct transformation functions change the schema structure of the DataFrame. These transformations integrate with functions whose parameter structs require a certain schema.

- **subset_struct:** subset fields from a struct

  ```python
  from pyspark.sql import Row
  row_one = Row(Row(str_col='foo', int_col=1, bool_col=True))
  row_two = Row(Row(str_col='bar', int_col=2, bool_col=False))
  base_df = spark.createDataFrame([row_one, row_two], schema=['base_col'])
  subsetted_df = base_df.selectExpr("subset_struct(base_col, 'str_col', 'bool_col') as subsetted_col")
  ```

- **add_struct_fields:** append fields to a struct

  ```python
  added_df = base_df.selectExpr("add_struct_fields(base_col, 'float_col', 3.14, 'rev_str_col', reverse(base_col.str_col)) as added_col")
  ```

- **expand_struct:** explode a struct into columns

  ```python
  expanded_df = base_df.selectExpr("expand_struct(base_col)")
  ```

3.10.2 Spark ML transformations

Glow supports transformations between double arrays and Spark ML vectors for integration with machine learning libraries such as Spark’s machine learning library (MLlib).

- **array_to_dense_vector:** transform from an array to a dense vector

  ```python
  array_df = spark.createDataFrame([Row([1.0, 2.0, 3.0]), Row([4.1, 5.1, 6.1])], schema=['array_col'])
  dense_df = array_df.selectExpr("array_to_dense_vector(array_col) as dense_vector_col")
  ```
• `array_to_sparse_vector`: transform from an array to a sparse vector

```python
glow
sparse_df = array_df.selectExpr('array_to_sparse_vector(array_col) as sparse_vector_col')
glow
```

• `vector_to_array`: transform from a vector to a double array

```python
glow
from pyspark.ml.linalg import SparseVector
glow
glow
row_one = Row(vector_col=SparseVector(3, [0, 2], [1.0, 3.0]))
glow
glow
row_two = Row(vector_col=SparseVector(3, [1], [1.0]))
glow
glow
vector_df = spark.createDataFrame([row_one, row_two])
glow
glow
array_df = vector_df.selectExpr('vector_to_array(vector_col) as array_col')
glow
```

• `explode_matrix`: explode a Spark ML matrix such that each row becomes an array of doubles

```python
glow
from pyspark.ml.linalg import DenseMatrix
glow
glow
matrix_df = spark.createDataFrame(Row([DenseMatrix(2, 3, range(6))]), schema=['matrix_col'])
glow
glow
array_df = matrix_df.selectExpr('explode_matrix(matrix_col) as array_col')
glow
```

#### 3.10.3 Variant data transformations

Glow supports numeric transformations on variant data for downstream calculations, such as GWAS.

• `genotype_states`: create a numeric representation for each sample’s genotype data. This calculates the sum of the calls (or −1 if any calls are missing); the sum is equivalent to the number of alternate alleles for biallelic variants.

```python
glow
from pyspark.sql.types import *
glow
glow
missing_and_hom_ref = Row([Row(calls=[-1, 0]), Row(calls=[0, 0])])
glow
glow
het_and_hom_alt = Row([Row(calls=[0, 1]), Row(calls=[1, 1])])
glow
glow
calls_schema = StructField('calls', ArrayType(IntegerType()))
glow
glow
genotypes_schema = StructField('genotypes_col', ArrayType(StructType([calls_schema])))
glow
glow
genotypes_df = spark.createDataFrame([missing_and_hom_ref, het_and_hom_alt],
StructType([genotypes_schema]))
glow
glow
num_alt_alleles_df = genotypes_df.selectExpr('genotype_states(genotypes_col) as num_alt_alleles_col')
glow
```

• `hard_calls`: get hard calls from genotype probabilities. These are determined based on the number of alternate alleles for the variant, whether the probabilities are phased (true for haplotypes and false for genotypes), and a call threshold (if not provided, this defaults to 0.9). If no calls have a probability above the threshold, the call is set to −1.

```python
glow
unphased_above_threshold = Row(probabilities=[0.0, 0.0, 0.0, 1.0, 0.0, 0.0], num_alts=2, phased=False)
glow
glow
phased_below_threshold = Row(probabilities=[0.1, 0.9, 0.8, 0.2], num_alts=1, phased=True)
glow
glow
uncalled_df = spark.createDataFrame([unphased_above_threshold, phased_below_threshold])
glow
glow
hard_calls_df = uncalled_df.selectExpr('hard_calls(probabilities, num_alts, phased, 0.95) as calls')
glow
```

• `mean_substitute`: substitutes the missing values of a numeric array using the mean of the non-missing values. Any values that are NaN, null or equal to the missing value parameter are considered missing. If all values are missing, they are substituted with the missing value. If the missing value is not provided, this defaults to −1.
```python
unsubstituted_row = Row(unsubstituted_values=[float('nan'), None, -1.0, 0.0, 1.0, 2.0, 3.0])
unsubstituted_df = spark.createDataFrame([unsubstituted_row])
substituted_df = unsubstituted_df.selectExpr("mean_substitute(unsubstituted_values, -1.0) as substituted_values")
```
Perform population-scale statistical analyses of genetic variants.

### 4.1 Parallelizing Command-Line Bioinformatics Tools With the Pipe Transformer

To use single-node tools on massive data sets, Glow includes a utility called the Pipe Transformer to process Spark DataFrames with command-line tools.

#### 4.1.1 Usage

Consider a minimal case with a DataFrame containing a single column of strings. You can use the Pipe Transformer to reverse each of the strings in the input DataFrame using the `rev` Linux command:

**Python**

```python
# Create a text-only DataFrame
df = spark.createDataFrame([['foo'], ['bar'], ['baz']], ['value'])
rev_df = glow.transform('pipe', df, cmd=['rev'], input_formatter='text', output_formatter='text')
```

**Scala**

Provide options as a `Map[String, Any].

```scala
Glow.transform("pipe", df, Map(
  "cmd" -> Seq("grep", "-v", "#INFO"),
  "inputFormatter" -> "text",
  "outputFormatter" -> "text",
  "inVcfHeader" -> "infer")
```

The options in this example demonstrate how to control the basic behavior of the transformer:

- `cmd` is a JSON-encoded array that contains the command to invoke the program
- `input_formatter` defines how each input row should be passed to the program
- `output_formatter` defines how the program output should be converted into a new DataFrame

The input DataFrame can come from any Spark data source — Delta, Parquet, VCF, BGEN, and so on.
4.1.2 Integrating with bioinformatics tools

To integrate with tools for genomic data, you can configure the Pipe Transformer to write each partition of the input DataFrame as VCF by choosing vcf as the input and output formatter. Here is an example using bedtools, note that the bioinformatics tool must be installed on each virtual machine of the Spark cluster.

```python
df = spark.read.format("vcf").load(path)
intersection_df = glow.transform('pipe',
    df,
    cmd=['bedtools', 'intersect', '-a', 'stdin', '-b', bed, '-header', '-wa'],
    input_formatter='vcf',
    in_vcf_header='infer',
    output_formatter='vcf'
)
```

You must specify a method to determine the VCF header when using the VCF input formatter. The option infer instructs the Pipe Transformer to derive a VCF header from the DataFrame schema. Alternately, you can provide the header as a blob, or you can point to the filesystem path for an existing VCF file with the correct header.

4.1.3 Options

Option keys and values are always strings. You can specify option names in snake or camel case; for example `inputFormatter`, `input_formatter`, and `InputFormatter` are all equivalent.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmd</td>
<td>The command, specified as an array of strings, to invoke the piped program. The program’s stdin receives the formatted contents of the input DataFrame, and the output DataFrame is constructed from its stdout. The stderr stream will appear in the executor logs.</td>
</tr>
<tr>
<td>input_formatter</td>
<td>Converts the input DataFrame to a format that the piped program understands. Built-in input formatters are text, csv, and vcf.</td>
</tr>
<tr>
<td>output_formatter</td>
<td>Converts the output of the piped program back into a DataFrame. Built-in output formatters are text, csv, and vcf.</td>
</tr>
<tr>
<td>env_*</td>
<td>Options beginning with env_ are interpreted as environment variables. Like other options, the environment variable name is converted to lower snake case. For example, providing the option env_aniMal=MONKEY results in an environment variable with key ani_mal and value MONKEY being provided to the piped program.</td>
</tr>
</tbody>
</table>

Some of the input and output formatters take additional options.
### VCF input formatter

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in_vcf_header</td>
<td>How to determine a VCF header from the input DataFrame. Possible values:</td>
</tr>
<tr>
<td></td>
<td>• <em>infer</em>: Derive a VCF header from the DataFrame schema. The inference behavior matches that of the <em>sharded VCF writer</em>.</td>
</tr>
<tr>
<td></td>
<td>• The complete contents of a VCF header starting with <code>##</code></td>
</tr>
<tr>
<td></td>
<td>• A Hadoop filesystem path to a VCF file. The header from this file is used as the VCF header for each partition.</td>
</tr>
</tbody>
</table>

The CSV input and output formatters accept most of the same options as the CSV data source. You must prefix options to the input formatter with `in_`, and options to the output formatter with `out_`. For example, `in_quote` sets the quote character when writing the input DataFrame to the piped program.

The following options are not supported:

- `path` options are ignored
- The `parserLib` option is ignored. `univocity` is always used as the CSV parsing library.

### 4.1.4 Cleanup

The pipe transformer uses RDD caching to optimize performance. Spark automatically drops old data partitions in a least-recently-used (LRU) fashion. If you would like to manually clean up the RDDs cached by the pipe transformer instead of waiting for them to fall out of the cache, use the pipe cleanup transformer on any DataFrame. Do not perform cleanup until the pipe transformer results have been materialized, such as by being written to a Delta Lake table.

**Python**

```python
import glow

glow.transform('pipe_cleanup', df)
```

**Scala**

```scala
import Glow

Glow.transform("pipe_cleanup", df)
```

### Notebook

<iframe src="../_static/notebooks/tertiary/pipe-transformer.html" id="90596753629376657" height="1000px" width="100%" style="overflow-y:visible;" scrolling="no"></iframe>
4.2 Using Python Statistics Libraries

This notebook demonstrates how to use pandas user-defined functions (UDFs) to run native Python code with PySpark when working with genomic data.

4.2.1 pandas example notebook

```python
# Read in VCF file
variants = spark.read.format('vcf').load(genotypes_vcf)

# genotype_states returns the number of alt alleles for each sample
# mean_substitute replaces any missing genotype states with the mean of the non-
# missing states
genotypes = glow.transform('split_multiallelics', variants) \
    .withColumn('gt', glow.mean_substitute(glow.genotype_states(col('genotypes')))) \
    .cache()

# Read covariates from a CSV file and add an intercept
covariates = pd.read_csv(covariates_csv, index_col=0)
covariates['intercept'] = 1.
```

4.3 Genome-wide Association Study Regression Tests

Glow contains functions for performing simple regression analyses used in genome-wide association studies (GWAS).

Tip: Glow automatically converts literal one-dimensional and two-dimensional numpy ndarrays of doubles to array<double> and spark.ml.DenseMatrix respectively.

4.3.1 Linear regression

linear_regression_gwas performs a linear regression association test optimized for performance in a GWAS setting.

Example

```python
from glow.wgr.functions import reshape_for_gwas
import numpy as np
import pandas as pd
from pyspark.ml.linalg import DenseMatrix
from pyspark.sql import Row
from pyspark.sql.functions import col, lit

# Read in VCF file
variants = spark.read.format('vcf').load(genotypes_vcf)

# genotype_states returns the number of alt alleles for each sample
# mean_substitute replaces any missing genotype states with the mean of the non-
# missing states
genotypes = glow.transform('split_multiallelics', variants) \
    .withColumn('gt', glow.mean_substitute(glow.genotype_states(col('genotypes')))) \
    .cache()

# Read covariates from a CSV file and add an intercept
covariates = pd.read_csv(covariates_csv, index_col=0)
covariates['intercept'] = 1.
```

(continues on next page)
# Read phenotypes from a CSV file

def pd_phenotypes = pd.read_csv(continuous_phenotypes_csv, index_col=0)
phenotypes = reshape_for_gwas(spark, pd_phenotypes)

# Run linear regression test
lin_reg_df = genotypes.crossJoin(phenotypes).select(
    'contigName',
    'start',
    'names',
    'label',
    glow.expand_struct(glow.linear_regression_gwas(
        col('gt'),
        phenotypes.values,
        lit(covariates.to_numpy())
    )
)

## Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>genotypes</td>
<td>array&lt;double&gt; (or numeric type that can be cast to double)</td>
<td>A numeric representation of the genotype for each sample at a given site, for example the result of the genotype_states function. This parameter can vary for each row in the dataset.</td>
</tr>
<tr>
<td>covariates</td>
<td>spark.ml Matrix</td>
<td>A matrix containing the covariates to use in the linear regression model. Each row in the matrix represents observations for a sample. The indexing must match that of the genotypes array that is, the 0th row in the covariate matrix should correspond to the same sample as the 0th element in the genotypes array. This matrix must be constant for each row in the dataset. If desired, you must explicitly include an intercept covariate in this matrix.</td>
</tr>
<tr>
<td>phenotypes</td>
<td>array&lt;double&gt; (or numeric type that can be cast to double)</td>
<td>A numeric representation of the phenotype for each sample. This parameter may vary for each row in the dataset. The indexing of this array must match the genotypes and covariates parameters.</td>
</tr>
</tbody>
</table>

## Return

The function returns a struct with the following fields. The computation of each value matches the `lm` R package.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>beta</td>
<td>double</td>
<td>The fit effect coefficient of the genotypes parameter.</td>
</tr>
<tr>
<td>standardError</td>
<td>double</td>
<td>The standard error of beta.</td>
</tr>
<tr>
<td>pValue</td>
<td>double</td>
<td>The P-value of the t-statistic for beta.</td>
</tr>
</tbody>
</table>

## Implementation details

The linear regression model is fit using the QR decomposition. For performance, the QR decomposition of the covariate matrix is computed once and reused for each (genotypes, phenotypes) pair.
4.3.2 Logistic regression

logistic_regression_gwas performs a logistic regression hypothesis test optimized for performance in a GWAS setting.

Example

```python
# Read a single phenotype from a CSV file
trait = 'Binary_Trait_1'
phenotype = np.hstack(pd.read_csv(binary_phenotypes_csv, index_col=0)[[trait]].to_numpy()).astype('double')

# Likelihood ratio test
lrt_log_reg_df = genotypes.select(
    'contigName',
    'start',
    'names',
    glow.expand_struct(glow.logistic_regression_gwas(
        col('gt'),
        lit(phenotype),
        lit(covariates.to_numpy()),
        'LRT'
    ))
)

# Firth test
firth_log_reg_df = genotypes.select(
    'contigName',
    'start',
    'names',
    glow.expand_struct(glow.logistic_regression_gwas(
        col('gt'),
        lit(phenotype),
        lit(covariates.to_numpy()),
        'Firth'
    ))
)
```

Parameters

The parameters for the logistic regression test are largely the same as those for linear regression. The primary differences are that the phenotypes values should be in the set \([0,1]\) and that there is one additional parameter test to specify the hypothesis test method.
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>genotypes</td>
<td>array&lt;double&gt; (or numeric type that can be cast to double)</td>
<td>A numeric representation of the genotype for each sample at a given site, for example the result of the <code>genotype_states</code> function. This parameter can vary for each row in the dataset.</td>
</tr>
<tr>
<td>covariates</td>
<td>spark.ml Matrix</td>
<td>A matrix containing the covariates to use in the logistic regression model. Each row in the matrix represents observations for a sample. The indexing must match that of the <code>genotypes</code> array that is, the 0th row in the covariate matrix should correspond to the same sample as the 0th element in the <code>genotypes</code> array. This matrix must be constant for each row in the dataset. If desired, you must explicitly include an intercept covariate in this matrix.</td>
</tr>
<tr>
<td>phenotypes</td>
<td>array&lt;double&gt; (or numeric type that can be cast to double)</td>
<td>A numeric representation of the phenotype for each sample. This parameter may vary for each row in the dataset. The indexing of this array must match the <code>genotypes</code> and <code>covariates</code> parameters.</td>
</tr>
<tr>
<td>test</td>
<td>string</td>
<td>The hypothesis test method to use. Currently likelihood ratio (LRT) and Firth (Firth) tests are supported.</td>
</tr>
</tbody>
</table>

**Return**

The function returns a struct with the following fields. The computation of each value matches the `glm` R package for the likelihood ratio test and the `logistf` R package for the Firth test.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>beta</td>
<td>double</td>
<td>Log-odds associated with the <code>genotypes</code> parameter, NaN if the fit failed.</td>
</tr>
<tr>
<td>oddsRatio</td>
<td>double</td>
<td>Odds ratio associated with the <code>genotypes</code> parameter, NaN if the fit failed.</td>
</tr>
<tr>
<td>waldConfidenceInterval</td>
<td>interval&lt;double&gt;</td>
<td>Wald 95% confidence interval of the odds ratio, NaN s if the fit failed.</td>
</tr>
<tr>
<td>pValue</td>
<td>double</td>
<td>p-value for the specified test. For the Firth test, this value is computed using the profile likelihood method. NaN if the fit failed.</td>
</tr>
</tbody>
</table>

**Implementation details**

The logistic regression null model and fully-specified model are fit using Newton iterations. For performance, the null model is computed once for each `phenotype` and used as a prior for each `(genotypes, phenotypes)` pair.

**Example notebook and blog post**

A detailed example and explanation of a GWAS workflow is available here.

**GWAS notebook**

How to run a notebook Get notebook link

4.3. Genome-wide Association Study Regression Tests
4.4 GloWGR: Whole Genome Regression

Glow supports Whole Genome Regression (WGR) as GloWGR, a parallelized version of the regenie method (see the preprint).

GloWGR consists of the following stages.

- Block the genotype matrix across samples and variants.
- Perform dimensionality reduction with ridge regression.
- Estimate phenotypic values with ridge regression.

Note: GloWGR currently supports only quantitative phenotypes.

4.4.1 Data preparation

GloWGR accepts three input datasources.
Genotype data

The genotype data may be read from any variant datasource supported by Glow, such as one read from VCF, BGEN or PLINK. For scalability, recommend ingesting flat genotype files into Delta tables.

The DataFrame must also include a column values containing a numeric representation of each genotype. The genotypic values may not be missing, or equal for every sample in a variant (eg. all samples are homozygous reference).

Example

When loading in the variants, perform the following transformations:

- Split multiallelic variants with the split_multiallelics transformer.
- Calculate the number of alternate alleles for biallelic variants with glow.genotype_states.
- Replace any missing values with the mean of the non-missing values using glow.mean_substitute.

```python
from pyspark.sql.functions import col, lit

variants = spark.read.format('vcf').load(genotypes_vcf)
genotypes = glow.transform('split_multiallelics', variants) 
    .withColumn('values', glow.mean_substitute(glow.genotype_states(col('genotypes'))))
```

Phenotype data

The phenotype data is represented as a Pandas DataFrame indexed by the sample ID. Each column represents a single phenotype. It is assumed that there are no missing phenotype values, and that the phenotypes are standardized with zero mean and unit (unbiased) standard deviation.

Example

```python
import pandas as pd

label_df = pd.read_csv(continuous_phenotypes_csv, index_col='sample_id')
label_df = label_df.fillna(label_df.mean())
label_df = ((label_df - label_df.mean())/label_df.std())[['Continuous_Trait_1', 'Continuous_Trait_2']]
```

Covariate data

The covariate data is represented as a Pandas DataFrame indexed by the sample ID. Each column represents a single covariate. It is assumed that there are no missing covariate values, and that the covariates are standardized with zero mean and unit (unbiased) standard deviation.

Example

```python
covariates = pd.read_csv(covariates_csv, index_col='sample_id')
covariates = covariates.fillna(covariates.mean())
covariates = (covariates - covariates.mean()) / covariates.std()
```
4.4.2 Genotype matrix blocking

glow.wgr.functions.block_variants_and_samples creates two objects: a block genotype matrix and a sample block mapping.

Parameters

- genotypes: Genotype DataFrame created by reading from any variant datasource supported by Glow, such as VCF. Must also include a column values containing a numeric representation of each genotype.
- sample_ids: List of sample IDs. Can be created by applying glow.wgr.functions.get_sample_ids to a genotype DataFrame.
- variants_per_block: Number of variants to include per block. We recommend 1000.
- sample_block_count: Number of sample blocks to create. We recommend 10.

Return

The function returns a block genotype matrix and a sample block mapping.

Warning: Variant rows in the input DataFrame whose genotype values are uniform across all samples are filtered from the output block genotype matrix.

Block genotype matrix

If we imagine the block genotype matrix conceptually, we think of an $N \times M$ matrix $X$ where each row $n$ represents an individual sample, each column $m$ represents a variant, and each cell $(n, m)$ contains a genotype value for sample $n$ at variant $m$. We then imagine laying a coarse grid on top of this matrix such that matrix cells within the same coarse grid cell are all assigned to the same block $x$. Each block $x$ is indexed by a sample block ID (corresponding to a list of rows belonging to the block) and a header block ID (corresponding to a list of columns belonging to the block). The sample block IDs are generally just integers 0 through the number of sample blocks. The header block IDs are strings of the form ‘chr_C_block_B’, which refers to the Bth block on chromosome C. The Spark DataFrame representing this block matrix can be thought of as the transpose of each block $x^T$ all stacked one atop another. Each row represents the values from a particular column from $X$, for the samples corresponding to a particular sample block. The fields in the DataFrame are:

- header: A column name in the conceptual matrix $X$.
- size: The number of individuals in the sample block for the row.
- values: Genotype values for this header in this sample block. If the matrix is sparse, contains only non-zero values.
- header_block: An ID assigned to the block $x$ containing this header.
- sample_block: An ID assigned to the block $x$ containing the group of samples represented on this row.
- position: An integer assigned to this header that specifies the correct sort order for the headers in this block.
- mu: The mean of the genotype calls for this header.
- sig: The standard deviation of the genotype calls for this header.
Sample block mapping

The sample block mapping consists of key-value pairs, where each key is a sample block ID and each value is a list of sample IDs contained in that sample block.

The order of these IDs match the order of the values arrays in the block genotype DataFrame.

Example

```python
from glow.wgr.linear_model import RidgeReducer, RidgeRegression
from glow.wgr.functions import block_variants_and_samples, get_sample_ids
from pyspark.sql.functions import col, lit

variants_per_block = 1000
sample_block_count = 10
sample_ids = get_sample_ids(genotypes)
block_df, sample_blocks = block_variants_and_samples(genotypes, sample_ids, variants_per_block, sample_block_count)
```

4.4.3 Dimensionality reduction

The first step in the fitting procedure is to apply a dimensionality reduction to the block matrix $X$ using the `RidgeReducer`.

This is accomplished by fitting multiple ridge models within each block $x$ and producing a new block matrix where each column represents the prediction of one ridge model applied within one block. This approach to model building is generally referred to as stacking. We will call the block genotype matrix we started with the level 0 matrix in the stack $X_0$, and the output of the ridge reduction step the level 1 matrix $X_1$. The `RidgeReducer` class is used for this step, which is initialized with a list of ridge regularization values (referred to here as alpha). Since ridge models are indexed by these alpha values, the `RidgeReducer` will generate one ridge model per value of alpha provided, which in turn will produce one column per block in $X_0$, so the final dimensions of matrix $X_1$ will be $N \times (L \times K)$, where $L$ is the number of header blocks in $X_0$ and $K$ is the number of alpha values provided to the `RidgeReducer`. In practice, we can estimate a span of alpha values in a reasonable order of magnitude based on guesses at the heritability of the phenotype we are fitting.

Initialization

When the `RidgeReducer` is initialized, it will assign names to the provided alphas and store them in a dictionary accessible as `RidgeReducer.alphas`.

Example

If alpha values are not provided, they will be generated during `RidgeReducer.fit` based on the unique number of headers $h$ in the blocked genotype matrix $X_0$, and a set of heritability values. These are only sensible if the phenotypes are on the scale of one.

$$\vec{\alpha} = h / [0.01, 0.25, 0.50, 0.75, 0.99]$$

```python
reducer = RidgeReducer()
```
Model fitting

In explicit terms, the reduction of a block $x_0$ from $X_0$ to the corresponding block $x_1$ from $X_1$ is accomplished by the matrix multiplication $x_0 \times B = x_1$, where $B$ is a coefficient matrix of size $m \times K$, where $m$ is the number of columns in block $x_0$ and $K$ is the number of alpha values used in the reduction. As an added wrinkle, if the ridge reduction is being performed against multiple phenotypes at once, each phenotype will have its own $B$, and for convenience we panel these next to each other in the output into a single matrix, so $B$ in that case has dimensions $m \times (K \times P)$ where $P$ is the number of phenotypes. Each matrix $B$ is specific to a particular block in $X_0$, so the Spark DataFrame produced by the `RidgeReducer` can be thought of all of as the matrices $B$ from all of the blocks stacked one atop another.

Parameters

- `block_df`: Spark DataFrame representing the beginning block matrix.
- `label_df`: Pandas DataFrame containing the target labels used in fitting the ridge models.
- `sample_blocks`: Dictionary containing a mapping of sample block IDs to a list of corresponding sample IDs.
- `covariates`: Pandas DataFrame containing covariates to be included in every model in the stacking ensemble (optional).

Return

The fields in the model DataFrame are:

- `header_block`: An ID assigned to the header block $x_0$ corresponding to the coefficients in this row.
- `sample_block`: An ID assigned to the sample block $x_0$ corresponding to the coefficients in this row.
- `header`: The name of a column from the conceptual matrix $X_0$ that correspond with a particular row from the coefficient matrix $B$.
- `alphas`: List of alpha names corresponding to the columns of $B$.
- `labels`: List of label (i.e., phenotypes) corresponding to the columns of $B$.
- `coefficients`: List of the actual values from a row in $B$.

Model transformation

After fitting, the `RidgeReducer.transform` method can be used to generate $X_1$ from $X_0$.

Parameters

- `block_df`: Spark DataFrame representing the beginning block matrix.
- `label_df`: Pandas DataFrame containing the target labels used in fitting the ridge models.
- `sample_blocks`: Dictionary containing a mapping of sample block IDs to a list of corresponding sample IDs.
- `model_df`: Spark DataFrame produced by the `RidgeReducer` fit method, representing the reducer model.
- `covariates`: Pandas DataFrame containing covariates to be included in every model in the stacking ensemble (optional).
Return

The output of the transformation is closely analogous to the block matrix DataFrame we started with. The main difference is that, rather than representing a single block matrix, it really represents multiple block matrices, with one such matrix per label (phenotype). Comparing the schema of this block matrix DataFrame (reduced_block_df) with the DataFrame we started with (block_df), the new columns are:

- **alpha**: This is the name of the alpha value used in fitting the model that produced the values in this row.
- **label**: This is the label corresponding to the values in this row. Since the genotype block matrix X0 is phenotype-agnostic, the rows in block_df were not restricted to any label/phenotype, but the level 1 block matrix X1 represents ridge model predictions for the labels the reducer was fit with, so each row is associated with a specific label.

The headers in the X1 block matrix are derived from a combination of the source block in X0, the alpha value used in fitting the ridge model, and the label they were fit with. These headers are assigned to header blocks that correspond to the chromosome of the source block in X0.

Example

Use the `fit_transform` function if the block genotype matrix, phenotype DataFrame, sample block mapping, and covariates are constant for both the model fitting and transformation.

```python
reduced_block_df = reducer.fit_transform(block_df, label_df, sample_blocks, ... covariates)
```

4.4.4 Estimate phenotypic values

The block matrix X1 can be used to fit a final predictive model that can generate phenotype predictions \( y\_hat \) using the `RidgeRegression` class.

Initialization

As with the `RidgeReducer` class, this class is initialized with a list of alpha values.

Example

If alpha values are not provided, they will be generated during `RidgeRegression.fit` based on the unique number of headers \( h \) in the blocked genotype matrix X1, and a set of heritability values. These are only sensible if the phenotypes are on the scale of one.

\[
\vec{\alpha} = h/[0.01, 0.25, 0.50, 0.75, 0.99]
\]

```python
regression = RidgeRegression()
```

Model fitting

This works much in the same way as the ridge reducer fitting, except that it returns two DataFrames.
Parameters

- `block_df`: Spark DataFrame representing the reduced block matrix.
- `label_df`: Pandas DataFrame containing the target labels used in fitting the ridge models.
- `sample_blocks`: Dictionary containing a mapping of sample block IDs to a list of corresponding sample IDs.
- `covariates`: Pandas DataFrame containing covariates to be included in every model in the stacking ensemble (optional).

Return

The first output is a model DataFrame analogous to the model DataFrame provided by the RidgeReducer. An important difference is that the header block ID for all rows will be 'all', indicating that all headers from all blocks have been used in a single fit, rather than fitting within blocks.

The second output is a cross validation report DataFrame, which reports the results of the hyperparameter (i.e., alpha) value optimization routine.

- `label`: This is the label corresponding to the cross cv results on the row.
- `alpha`: The name of the optimal alpha value
- `r2_mean`: The mean out of fold r2 score for the optimal alpha value

Model transformation

After fitting the RidgeRegression model, the model DataFrame and cross validation DataFrame are used to apply the model to the block matrix DataFrame to produce predictions (`y_hat`) for each label and sample using the RidgeRegression.transform or RidgeRegression.transform_loco method. We describe the leave-one-chromosome-out (LOCO) approach.

Parameters

- `block_df`: Spark DataFrame representing the reduced block matrix.
- `label_df`: Pandas DataFrame containing the target labels used in fitting the ridge models.
- `sample_blocks`: Dictionary containing a mapping of sample block IDs to a list of corresponding sample IDs.
- `model_df`: Spark DataFrame produced by the RidgeRegression.fit method, representing the reducer model
- `cv_df`: Spark DataFrame produced by the RidgeRegression.fit method, containing the results of the cross validation routine.
- `covariates`: Pandas DataFrame containing covariates to be included in every model in the stacking ensemble (optional).
- `chromosomes`: List of chromosomes for which to generate a prediction (optional). If not provided, the chromosomes will be inferred from the block matrix.
Return

The resulting $y_{hat}$ Pandas DataFrame is shaped like $label\_df$, indexed by the sample ID and chromosome with each column representing a single phenotype.

Example

```python
model\_df, cv\_df = regression.fit(reduced\_block\_df, label\_df, sample\_blocks, covariates)
y\_hat\_df = regression.transform\_loco(reduced\_block\_df, label\_df, sample\_blocks, model\_df, cv\_df, covariates)
```

Example notebook

GloWGR notebook

<a href="../additional-resources.html#running-databricks-notebooks">How to run a notebook</a>  
<a style='float:right' href="../_static/notebooks/tertiary/glowgr.html">Get notebook link</a>
TROUBLESHOOTING

- Job is slow or OOMs (throws an OutOfMemoryError) while using an aggregate like `collect_list` or `sample_call_summary_stats`
  - Try disabling the ObjectHashAggregate by setting `spark.sql.execution.useObjectHashAggregateExec` to `false`
- Job is slow or OOMs while writing to partitioned table
  - This error can occur when reading from highly compressed files. Try decreasing `spark.files.maxPartitionBytes` to a smaller value like 33554432 (32MB)
- My VCF looks weird after merging VCFs and saving with `bigvcf`
  - When saving to a VCF, the samples in the genotypes array must be in the same order for each row. This ordering is not guaranteed when using `collect_list` to join multiple VCFs. Try sorting the array using `sort_array`. 
6.1 Introducing GloWGR: An industrial-scale, ultra-fast and sensitive method for genetic association studies

Authors: Leland Barnard, Henry Davidge, Karen Feng, Joelle Mbatchou, Boris Boutkov, Kiavash Kianfar, Lukas Habegger, Jonathan Marchini, Jeffrey Reid, Evan Maxwell, Frank Austin Nothaft

June 22, 2020

The industry partnership between Regeneron and Databricks is enabling innovations in genomics data processing and analysis. Today, we announce that we are making a new whole genome regression method available to the open source bioinformatics community as part of Project Glow.

Large cohorts of individuals with paired clinical and genome sequence data enable unprecedented insight into human disease biology. Population studies such as the UK Biobank, Genomics England, or Genome Asia 100k datasets are driving a need for innovation in methods for working with genetic data. These methods include genome wide association studies (GWAS), which enrich our understanding of the genetic architecture of the disease and are used in cutting-edge industrial applications, such as identifying therapeutic targets for drug development. However, these datasets pose novel statistical and engineering challenges. The statistical challenges have been addressed by tools such as SAIGE and Bolt-LMM, but they are difficult to set up and prohibitively slow to run on biobank-scale datasets.

In a typical GWAS, a single phenotype such as cholesterol levels or diabetes diagnosis status is tested for statistical association with millions of genetic variants across the genome. Sophisticated mixed model and whole genome regression-based approaches have been developed to control for relatedness and population structure inherent to large genetic study populations when testing for genetic associations; several methods such as BOLT-LMM, SAIGE, and fastGWA use a technique called whole genome regression to sensitively analyze a single phenotype in biobank-scale projects. However, deeply phenotyped biobank-scale projects can require tens of thousands of separate GWAs to analyze the full spectrum of clinical variables, and current tools are still prohibitively expensive to run at scale. In order to address the challenge of efficiently analyzing such datasets, the Regeneron Genetics Center has just developed a new approach for the whole-genome regression method that enables running GWAS across upwards of hundreds of phenotypes simultaneously. This exciting new tool provides the same superior test power as current state-of-the-art methods at a small fraction of the computational cost.

This new whole genome regression (WGR) approach recasts the whole genome regression problem to an ensemble model of many small, genetic region-specific models. This method is described in a preprint released today, and implemented in the C++ tool regenie. As part of the collaboration between the Regeneron Genetics Center and Databricks on the open source Project Glow, we are excited to announce GloWGR, a lightning-fast and highly scalable distributed implementation of this WGR algorithm, designed from the ground up with Apache Spark and integrated with other Glow functionality. With GloWGR, performing WGR analyses on dozens of phenotypes can be accomplished simultaneously in a matter of minutes, a task that would require hundreds or thousands of hours with existing state-of-the-art tools. Moreover, GloWGR distributes along both the sample and genetic variant matrix dimensions, allowing for linear scaling and a high degree of data and task parallelism. GloWGR plugs seamlessly into any existing GWAS workflow, providing an immediate boost to association detection power at a negligible computational cost.
6.1.1 Achieving High Accuracy and Efficiency with Whole-Genome Regression

This whole genome regression tool has a number of virtues. First, it is more efficient: as implemented in the single node, open-source regenie tool, whole genome regression is orders of magnitude faster than either SAIGE, Bolt-LMM, or fastGW A, while producing equivalent results (Figure 1). Second, it is straightforward to parallelize: in the next section, we describe how we implemented whole genome regression using Apache Spark™ in the open-source Project Glow.

![Fig. 6.1: Comparison of GWAS results for three quantitative phenotypes from the UK Biobank project, produced by REGENIE, BOLT-LMM, and fastGW A.](image)

In addition to performance considerations, the whole genome regression approach produces covariates that are compatible with standard GWAS methods, and which eliminate spurious associations caused by population structure that are seen with traditional approaches. The Manhattan plots in figure 2 below compare the results of a traditional linear regression GWAS using standard covariates, to a linear regression GWAS using the covariates generated by WGR. This flexibility of GloWGR is another tremendous advantage over existing GWAS tools, and will allow for a wide variety of exciting extensions to the association testing framework that is already available in Glow.

Figure 3 shows performance comparisons between GloWGR, REGENIE, BoltLMM, and fastGW A. We benchmarked the whole genome regression test implemented in Glow against the C++ implementation available in the single-node regenie tool to validate the accuracy of the method. We found that the two approaches achieve statistically identical results. We also found that the Apache Spark™ based implementation in Glow scales linearly with the number of nodes used.

6.1.2 Scaling Whole Genome Regression within Project Glow

Performing WGR analysis with GloWGR has 5 steps:

- Dividing the genotype matrix into contiguous blocks of SNPs (~1000 SNPs per block, referred to as loci)
- Fitting multiple ridge models (~10) with varying ridge penalties within each locus
- Using the resulting ridge models to reduce the locus from a matrix of 1,000 features to 10 features (each feature is the prediction of one of the ridge models)
- Pooling the resulting features of all loci into a new reduced feature matrix \( X \) (\( N \) individuals by \( L \) loci x \( J \) ridge models per locus)
6.1. Introducing GloWGR: An industrial-scale, ultra-fast and sensitive method for genetic association studies

Fig. 6.2: Comparison of GWAS results of the quantitative phenotype bilirubin from the UK Biobank project, evaluated using standard linear regression and linear regression with GloWGR. The heightened peaks in the highlighted regions show the increase in power to detect subtler associations that is gained with GloWGR.

Fig. 6.3: Left: end-to-end GWAS runtime comparison for 50 quantitative traits from the UK Biobank project. Right: Run time comparison to fit WGR models against 50 quantitative phenotypes from the UK Biobank project. GloWGR scales well with cluster size, allowing for modeling of dozens of phenotypes in minutes without costing additional CPU efficiency. The exact list of phenotypes and computation environment details can be found here.
• Fitting a final regularized model from $X$ for the genome-wide contribution to phenotype $Y$.

Glow provides the easy-to-use abstractions shown in figure 4 for transforming large genotype matrices into the blocked matrix (below, left) and then fitting the whole genome regression model (below, right). These can be applied to data loaded in any of the genotype file formats that Glow understands, including VCF, Plink, and BGEN formats, as well as genotype data stored in Apache Spark™ native file formats like Delta Lake.

![Fig. 6.4: Creating a matrix grouped by locus and fitting mixed ridge regression models using GloWGR](image)

Glow provides an implementation of the WGR method for quantitative traits, and a binary trait variant is in progress. The covariate-adjusted phenotype created by GloWGR can be written out as an Apache Parquet™ or Delta Lake dataset, which can easily be loaded by and analyzed within Apache Spark, pandas, and other tools. Ultimately, using the covariates computed with WGR in a genome-wide association study is as simple as running the command shown in Figure 5, below. This command is run by Apache Spark™, in parallel, across all of the genetic markers under test.

```python
1 block_df, sample_blocks = block_variants_and_samples(variant_df,
2 sample_ids, variants_per_block, sample_block_count)
```

![Fig. 6.5: Updating phenotypes with the WGR results and running a GWAS using the built-in association test methods from Glow](image)

6.1.3 Join us and try whole genome regression in Glow!

Whole genome regression is available in Glow, which is an open source project hosted on Github, with an Apache 2 license. You can get started with this notebook that shows how to use GloWGR on data from 1,000 Genomes, by reading the preprint, by reading our project docs, or you can create a fork of the repository to start contributing code today. Glow is installed in the Databricks Genomics Runtime (Azure | AWS), and you can start a preview today.
6.2 Glow 0.4 Enables Integration of Genomic Variant and Annotation Data

Author: Kiavash Kianfar
June 9, 2020

Glow 0.4 was released on May 20, 2020. This blog focuses on the highlight of this release, the newly introduced capability to ingest genomic annotation data from the GFF3 (Generic Feature Format Version 3) flat file format. This release also includes other feature and usability improvements, which will be briefly reviewed at the end of this blog.

GFF3 is a sequence annotation flat file format proposed by the Sequence Ontology Project in 2013, which since has become the de facto format for genome annotation and is widely used by genome browsers and databases such as NCBI RefSeq and GenBank. GFF3, a 9-column tab-separated text format, typically carries the majority of the annotation data in the ninth column, called attributes, as a semi-colon-separated list of <tag>=<value> entries. As a result, although GFF3 files can be read as Spark DataFrames using Spark SQL’s standard csv data source, the schema of the resulting DataFrame would be quite unwieldy for query and data manipulation of annotation data, because the whole list of attribute tag-value pairs for each sequence will appear as a single semi-colon-separated string in the attributes column of the DataFrame.

Glow 0.4 adds the new and flexible gff Spark SQL data source to address this challenge and create a smooth GFF3 ingest and query experience. While reading the GFF3 file, the gff data source parses the attributes column of the file to create an appropriately typed column for each tag. In each row, this column will contain the value corresponding to that tag in that row (or null if the tag does not appear in the row). Consequently, all tags in the GFF3 attributes column will have their own corresponding column in the Spark DataFrame, making annotation data query and manipulation much easier.

6.2.1 Ingesting GFF3 Annotation Data

Like any other Spark data source, reading GFF3 files using Glow’s gff data source can be done in a single line of code. As an example, we can ingest the annotations of the Homo Sapiens genome assembly GRCh38.p13 from a GFF3 file (obtained from RefSeq ftp site) as shown below. Here, we have also filtered the annotations to chromosome 22 in order to use the resulting annotations_df DataFrame (Fig. 6.6) in continuation of our example. The annotations_df alias is for the same purpose as well.

```python
import glow
glow.register(spark)

# GFF3 file path

gff_path = r'/databricks-datasets/genomics/gffs/GCF_000001405.39_GRCh38.p13_genomic.

# Read GFF3 file using gff data source

annotations_df = spark.read.format('gff').load(gff_path) \
    .filter("seqid = 'NC_000022.11'") \
    .alias('annotations_df')
```

In addition to reading uncompressed .gff files, the gff data source supports all compression formats supported by Spark’s csv data source, including .gz and .bgz. It is strongly recommended to use splittable compression formats like .bgz instead of .gz for better parallelization of the read process.

6.2.2 Schema

Let us have a closer look at the schema of the resulting DataFrame, which was automatically inferred by Glow’s gff data source:
This schema has 100 fields (not all shown here). The first eight fields (seqId, source, type, start, end, score, strand, and phase), here referred to as the “base” fields, correspond to the first eight columns of the GFF3 format cast in the proper data types. The rest of the fields in the inferred schema are the result of parsing the attributes column of the GFF3 file. Fields corresponding to any “official” tag (those referred to as “tags with pre-defined meaning” in the GFF3 format description), if present in the GFF3 file, come first in appropriate data types. The official fields are followed by the “unofficial” fields (fields corresponding to any other tag) in alphabetical order. In the example above, ID, Name, Parent, Target, Gap, Note, Dbxref, and Is_circular are the official fields, and the rest are the unofficial fields. The gff data source discards the comments, directives, and FASTA lines that may be in the GFF3 file.
As it is not uncommon for the official tags to be spelled differently in terms of letter case and underscore usage across different GFF3 files, or even within a single GFF3 file, the gff data source is designed to be insensitive to letter case and underscore in extracting official tags from the attributes field. For example, the official tag Dbxref will be correctly extracted as an official field even if it appears as dbxref or dbx_ref in the GFF3 file. Please see Glow documentation for more details.

Like other Spark SQL data sources, Glow’s gff data source is also able to accept a user-specified schema through the .schema command. The data source behavior in this case is also designed to be quite flexible. More specifically, the fields (and their types) in the user-specified schema are treated as the list of fields, whether base, official, or unofficial, to be extracted from the GFF3 file (and cast to the specified types). Please see the Glow documentation for more details on how user-specified schemas can be used.

6.2.3 Example: Gene Transcripts and Transcript Exons

With the annotation tags extracted as individual DataFrame columns using Glow’s gff data source, query and data preparation over genetic annotations becomes as easy as writing common Spark SQL commands in the user’s API of choice. As an example, here we demonstrate how simple queries can be used to extract data regarding hierarchical grouping of genomic features from the annotations_df created above.

One of the main advantages of the GFF3 format compared to older versions of GFF is the improved presentation of feature hierarchies (see GFF3 format description for more details). Two examples of such hierarchies are:

- Transcripts of a gene (here, gene is the “parent” feature and its transcripts are the “children” features).
- Exons of a transcript (here, the transcript is the parent and its exons are the children).

In the GFF3 format, the parents of the feature in each row are identified by the value of the parent tag in the attributes column, which includes the ID(s) of the parent(s) of the row. Glow’s gff data source extracts this information as an array of parent ID(s) in a column of the resulting DataFrame called parent.

Assume we would like to create a DataFrame, called gene_transcript_df, which, for each gene on chromosome 22, provides some basic information about the gene and all its transcripts. As each row in the annotations_df of our example has at most a single parent, the parent_child_df DataFrame created by the following query will help us in achieving our goal. This query joins annotations_df with a subset of its own columns on the parent column as the key. Fig. 6.7 shows a small section of parent_child_df.

```python
from pyspark.sql.functions import *

parent_child_df = annotations_df \
    .join(  
        annotations_df.select('id', 'type', 'name', 'start', 'end').alias('parent_df'),  
        col('annotations_df.parent')[0] == col('parent_df.id') # each row in annotation_df has at most one parent  
    )  \
    .orderBy('annotations_df.start', 'annotations_df.end')  \
    .select(  
        'annotations_df.seqid',  
        'annotations_df.type',  
        'annotations_df.start',  
        'annotations_df.end',  
        'annotations_df.id',  
        'annotations_df.name',  
        col('annotations_df.parent')[0].alias('parent_id'),  
        col('parent_df.Name').alias('parent_name'),  
        col('parent_df.type').alias('parent_type'),  
        col('parent_df.start').alias('parent_start'),  
        col('parent_df.end').alias('parent_end')  
    )
```

(continues on next page)
Having the `parent_child_df` DataFrame, we can now write the following simple function, called `parent_child_summary`, which, given this DataFrame, the parent type, and the child type, generates a DataFrame containing basic information on each parent of the given type and all its children of the given type.

```python
from pyspark.sql.dataframe import *

def parent_child_summary(parent_child_df: DataFrame, parent_type: str, child_type: str) -> DataFrame:
    return parent_child_df \\
    .select( \\
        'seqid',
        col('parent_id').alias(f'{parent_type}_id'),
        col('parent_name').alias(f'{parent_type}_name'),
        col('parent_start').alias(f'{parent_type}_start'),
        col('parent_end').alias(f'{parent_type}_end'),
        col('id').alias(f'{child_type}_id'),
        col('start').alias(f'{child_type}_start'),
        col('end').alias(f'{child_type}_end'),
    ) \\
    .where(f'{type} == "{child_type}" and {parent_type} == "{parent_type}"') \\
    .groupBy( \\
        'seqid',
        f'{parent_type}_id',
        f'{parent_type}_name',
        f'{parent_type}_start',
        f'{parent_type}_end',
    ) \\
    .agg( \\
        collect_list( \\
            struct( \\
                f'{child_type}_id',
                f'{child_type}_start',
                f'{child_type}_end',
            ) \\
        ).alias(f'{child_type}s'),
    ) \\
    .orderBy( \\
        f'{parent_type}_start',
    )
```

Fig. 6.7: A small section of the `parent_child_df` DataFrame

(continues on next page)
Now we can generate our intended `gene_transcript_df` DataFrame, shown in Fig. 6.8, with a single call to this function:

```python
gene_transcript_df = parent_child_summary(parent_child_df, 'gene', 'transcript')
```

Fig. 6.8: A small section of the `gene_transcript_df` DataFrame

In each row of this DataFrame, the `transcripts` column contains the ID, start and end of all transcripts of the gene in that row as an array of structs.

The same function can now be used to generate any parent-child feature summary. For example, we can generate the information of all exons of each transcript on chromosome 22 with another call to the `parent_child_summary` function as shown below. Fig. 6.9 shows the generated `transcript_exon_df` DataFrame.

```python
transcript_exon_df = parent_child_summary(parent_child_df, 'transcript', 'exon')
```

Fig. 6.9: A small section of the `transcript_exon_df` DataFrame

### 6.2.4 Example Continued: Integration with Variant Data

Glow has data sources to ingest variant data from common flat file formats such as VCF, BGEN, and PLINK. Combining the power of Glow’s variant data sources with the new gff data source, the users can now seamlessly annotate their variant DataFrames by joining them with annotation DataFrames in any desired fashion.

As an example, let us load the chromosome 22 variants of the 1000 Genome Project (on genome assembly GRCh38) from a VCF file (obtained from the project’s ftp site). Fig. 6.10 shows the resulting `variants_df`.
vcf_path = "~/databricks-datasets/genomics/1kg-vcfs/ALL.chr22.shapeit2_integrated_
    →snvindels_v2a_27022019.GRCh38.phased.vcf.gz"

variants_df = spark.read \
    .format("vcf") \
    .load(vcf_path) \
    .alias('variants_df')

Fig. 6.10: A small section of the variants_df DataFrame

Now using the following double-join query, we can create a DataFrame which, for each variant on a gene on chromosome 22, provides the information of the variant as well as the exon, transcript, and gene on which the variant resides (Fig. 6.11). Note that the first two exploded DataFrames can also be constructed directly from parent_child_df. Here, since we had already defined gene_transcript_df and transcript_exon_df, we generated these exploded DataFrames simply by applying the explode function followed by Glow’s expand_struct function on them.

```python
from glow.functions import *

gene_transcript_exploded_df = gene_transcript_df \
    .withColumn('transcripts', explode('transcripts')) \
    .withColumn('transcripts', expand_struct('transcripts')) \
    .alias('gene_transcript_exploded_df')

transcript_exon_exploded_df = transcript_exon_df \
    .withColumn('exons', explode('exons')) \
    .withColumn('exons', expand_struct('exons')) \
    .alias('transcript_exon_exploded_df')

variant_exon_transcript_gene_df = variants_df \
    .join( \
        transcript_exon_exploded_df, \
        (variants_df.start < transcript_exon_exploded_df.exon_end) & \
        (transcript_exon_exploded_df.exon_start < variants_df.end) \
    ) \
    .join( \
        gene_transcript_exploded_df, \
        transcript_exon_exploded_df.transcript_id == gene_transcript_exploded_df.transcript_id \
    ) \
    .select( \
        col('variants_df.contigName').alias('variant_contig'), \
        col('variants_df.start').alias('variant_start'), \
        col('variants_df.end').alias('variant_end'), \
        col('variants_df.referenceAllele'), \
        col('variants_df.alternateAlleles'), \
        'transcript_exon_exploded_df.exon_id', \
        'transcript_exon_exploded_df.exon_start', \
    )
```

(continues on next page)
In addition to the new gff reader, Glow 0.4 introduced other features and improvements. A new function, called `mean_substitute`, was introduced, which can be used to substitute the missing values of a numeric Spark array with the mean of the non-missing values. The `normalize_variants` transformer now accepts reference genomes in bgzipped fasta format in addition to the uncompressed fasta. The VCF reader was updated to be able to handle reading file globs that include tabix index files. In addition, this reader no longer has the `splitToBiallelic` option. The `split_multiallelics` transformer introduced in Glow 0.3 can be used instead. Also, the `pipe` transformer was improved so that it does not pipe empty partitions. As a result, users do not need to `repartition` or `coalesce` when piping VCF files. For a complete list of new features and improvements in Glow 0.4, please refer to Glow 0.4 Release Notes.

### 6.2.6 Try It!

Try Glow 0.4 and its new features here.
6.3 Glow 0.3.0 Introduces Several New Large-Scale Genomic Analysis Features

Author: Kiavash Kianfar
March 2, 2020

Glow 0.3.0 was released on February 21, 2020, improving Glow’s power and ease of use in performing large-scale genomic analysis. In this blog, we highlight features and improvements introduced in this release.

6.3.1 Python and Scala APIs for Glow SQL functions

In this release, Python and Scala APIs were introduced for all Glow SQL functions, similar to what is available for Spark SQL functions. In addition to improved simplicity, this provides enhanced compile-time safety. The SQL functions and their Python and Scala clients are generated from the same source so any new functionality in the future will always appear in all three languages. Please refer to *Glow PySpark Functions* for more information on Python APIs for these functions. As an example, the usage of such Python and Scala APIs for the function `normalize_variant` is presented at the end of next section.

6.3.2 Improved variant normalization

The variant normalizer received a major improvement in this release. It still behaves like `bcftools norm` and `vt normalize`, but is about 2.5x faster and has a more flexible API. Moreover, the new normalizer is implemented as a function in addition to a transformer.

`normalize_variants` transformer: The improved transformer preserves the columns of the input DataFrame, adds the normalization status to the DataFrame, and has the option of adding the normalization results (including the normalized coordinates and alleles) to the DataFrame as a new column. As an example, assume we read the `original_variants_df` DataFrame shown in Fig. 6.12 by issuing the following command:

```python
original_variants_df = spark.read \
    .format("vcf") \
    .option("includeSampleIds", False) \
    .load("/databricks-datasets/genomics/call-sets")
```

The output DataFrame of this improved transformer looks like Fig. 6.13. The `start`, `end`, `referenceAllele`, and `alternateAlleles` fields are updated with the normalized values and a `normalizationStatus` column is added to the DataFrame. This column contains a `changed` subfield indicating whether normalization changed the variant and an `errorMessage` subfield containing the error message in case of an error.

The newly introduced `replace_columns` option can be used to add the normalization results as a new column to the DataFrame instead of replacing the original `start`, `end`, `referenceAllele`, and `alternateAlleles` fields. This can be done as follows:
Fig. 6.12: The variant DataFrame original_variants_df

<table>
<thead>
<tr>
<th>contigName</th>
<th>start</th>
<th>end</th>
<th>referenceAllele</th>
<th>alternateAlleles</th>
<th>qual</th>
<th>filters</th>
<th>splitFromMultAllelicINFO_AK</th>
<th>INFO_AK</th>
<th>INFO_AC</th>
<th>genotypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>chr20</td>
<td>12385781</td>
<td>12385782</td>
<td>AAAAAA</td>
<td>AGATCATCCTTCCTTTT</td>
<td>30.6</td>
<td>false</td>
<td>2</td>
<td>0.25</td>
<td>1</td>
<td>false, 1, 1,...</td>
</tr>
<tr>
<td>chr20</td>
<td>12385499</td>
<td>12385499</td>
<td>AAAAA</td>
<td>GTGCTCTCTTCTCCT</td>
<td>30.6</td>
<td>false</td>
<td>2</td>
<td>0.55</td>
<td>2</td>
<td>false, 1, 1,...</td>
</tr>
<tr>
<td>chr20</td>
<td>12383234</td>
<td>12383234</td>
<td>AAAAAAAAAAAA</td>
<td>TCTCTCTCTTTTTT</td>
<td>30.6</td>
<td>false</td>
<td>2</td>
<td>0.25</td>
<td>1</td>
<td>false, 1, 1,...</td>
</tr>
<tr>
<td>chr20</td>
<td>12388338</td>
<td>12388339</td>
<td>GAG</td>
<td>GAGAGAGAGAGAGAGAGAGAG</td>
<td>30.6</td>
<td>false</td>
<td>2</td>
<td>0.55</td>
<td>2</td>
<td>false, 1, 1,...</td>
</tr>
<tr>
<td>chr20</td>
<td>12388396</td>
<td>12388396</td>
<td>CGGA</td>
<td>CGGA</td>
<td>30.6</td>
<td>false</td>
<td>2</td>
<td>0.55</td>
<td>2</td>
<td>false, 1, 1,...</td>
</tr>
<tr>
<td>chr20</td>
<td>12388341</td>
<td>12388342</td>
<td>AAAAA</td>
<td>AAAAA</td>
<td>30.6</td>
<td>false</td>
<td>2</td>
<td>0.55</td>
<td>2</td>
<td>false, 1, 1,...</td>
</tr>
</tbody>
</table>

Fig. 6.13: The normalized DataFrame normalized_variants_df
import glow
normalized_variants_df = glow.transform("normalize_variants",
    original_variants_df,
    replace_columns="False",
    reference_genome_path="/mnt/dbnucleus/dbgenomics/grch38/data/GRCh38_full_analysis_set_plus_decoy_hla.fa"
)

# Fig. 6.14: The normalized DataFrame normalized_noreplace_variants_df with normalization results added as a new column

The resulting DataFrame will be as shown in Fig. 6.14, where a normalizationResults column containing the normalized start, end, referenceAllele, alternateAlleles, and normalizationStatus subfields is added to the DataFrame.

We also note that since the multiallelic variant splitter is implemented as a separate transformer in this release (see below), the mode option of the normalize_variants transformer is deprecated. Refer to Variant Normalization for more details on the normalize_variants transformer.

to variants_sql

from pyspark.sql.functions import expr
function_normalized_variants_df = original_variants_df.withColumn("normalizationResult",

    expr("normalize_variant(contigName, start, end, referenceAllele, alternateAlleles,
    '/mnt/dbnucleus/dbgenomics/grch38/data/GRCh38_full_analysis_set_plus_decoy_hla.fa')")
)

As discussed in the previous section, this SQL expression function, like any other in Glow, now has Python and Scala APIs as well. Therefore, the same can be done in Python as follows:

from glow.functions import normalize_variant
function_normalized_variants_df = original_variants_df.withColumn("normalizationResult",

    normalize_variant(contigName, start, end, referenceAllele)
)

(continues on next page)
and in Scala as well, assuming `original_variant_df` is defined in Scala:

```scala
import io.projectglow.functions.normalize_variant
import org.apache.spark.sql.functions.col
val function_normalized_variants_df = original_variants_df.withColumn(  
  "normalizationResult",  
  normalize_variant(  
    col("contigName"),  
    col("start"),  
    col("end"),  
    col("referenceAllele"),  
    col("alternateAlleles"),  
    "/mnt/dbnucleus/dbgenomics/qrch38/data/GRCh38_full_analysis_set_plus_decoy_hla.fa"
  )
)
```

The result of any of the above commands will be the same as Fig. 6.14.

### 6.3.3 A new transformer to split multiallelic variants

This release also introduced a new DataFrame transformer, called `split_multiallelics`, to split multiallelic variants into biallelic ones with a behavior similar to vt decompose with --s option. This behavior is significantly more powerful than the behavior of the previous version’s splitter which behaved like GATK’s LeftAlignAndTrimVariants with --split-multi-allelics. In particular, the array-type INFO and genotype fields with elements corresponding to reference and alternate alleles are split “smart”ly (see --s option of vt decompose) into biallelic rows. So are the array-type genotype fields with elements sorted in colex order of genotype calles, e.g., the GL, PL, and GP fields in the VCF format. Moreover, an OLD_MULTIALLELIC INFO field is added to the DataFrame to store the original multiallelic form of the split variants.

The following is an example of using the `split_multiallelic` transformer on the `original_variants_df`.

```scala
import glow
split_variants_df = glow.transform("split_multiallelics", original_variants_df)
```

Please note that the new splitter is implemented as a separate transformer from the `normalize_variants` transformer. Previously, splitting could only be done as one of the operation modes of the `normalize_variants` transformer using the now-deprecated mode option.

Please refer to the documentation of the `split_multiallelics` transformer for complete details on the bahavior of this new transformer.

### 6.3.4 Parsing of Annotation Fields

The VCF reader and pipe transformer now parse variant annotations from tools such as SnpEff and VEP. This flattens the ANN and CSQ INFO fields, simplifying and accelerating queries on annotations. See the following query and its result in Fig. 6.16 for an example.
Fig. 6.15: The split DataFrame `split_variants_df`

```python
from pyspark.sql.functions import expr
variants_df = spark.read
  .format("vcf")
  .load("dbfs:/databricks-datasets/genomics/vcfs/loftee.vcf")
annotated_variants_df = original_variants_df.withColumn( 
    "Exploded_INFO_CSQ", 
    expr("explode(INFO_CSQ)") 
)
.selectExpr("contigName", "start", "end", "referenceAllele", "alternateAlleles", "expand_struct(Exploded_INFO_CSQ)", "genotypes")
```

Fig. 6.16: The annotated DataFrame `annotated_variants_df` with expanded subfields of the exploded INFO_CSQ

### 6.3.5 Other Improvements

Glow 0.3.0 also includes optimized implementations of the linear and logistic regression functions, resulting in ~50% performance improvements. See the documentation at [Linear regression](#) and [Logistic regression](#).

Furthermore, the new release supports Scala 2.12 in addition to Scala 2.11. The maven artifacts for both Scala versions are available on [Maven Central](#).
6.3.6 Try It!

Try Glow 0.3.0 and its new features here.

6.4 Streamlining Variant Normalization on Large Genomic Datasets

Author: Kiavash Kianfar
November 20, 2019

Many research and drug development projects in the genomics world involve large genomic variant data sets, the volume of which has been growing exponentially over the past decade. However, the tools to extract, transform, load (ETL) and analyze these data sets have not kept pace with this growth. Single-node command line tools or scripts are very inefficient in handling terabytes of genomics data in these projects. In October of this year, Databricks and the Regeneron Genetics Center partnered to introduce the open-source project Glow, which provides powerful genomics tools based on Apache Spark in order to address this issue.

In large cross-team research or drug discovery projects, computational biologists and bioinformaticians usually need to merge very large variant call sets in order to perform downstream analyses. In a prior post, we showcased the power and simplicity of Glow in ETL and merging of variant call sets from different sources using Glow’s VCF and BGEN Data Sources at unprecedented scales. Differently sourced variant call sets impose another major challenge. It is not uncommon for these sets to be generated by different variant calling tools and methods. Consequently, the same genomic variant may be represented differently (in terms of genomic position and alleles) across different call sets. These discrepancies in variant representation must be resolved before any further analysis on the data. This is critical for the following reasons:

1. To avoid incorrect bias in the results of downstream analysis on the merged set of variants or waste of analysis effort on seemingly new variants due to lack of normalization, which are in fact redundant (see Tan et al. for examples of this redundancy in 1000 Genome Project variant calls and dbSNP)
2. To ensure that the merged data set and its post-analysis derivations are compatible and comparable with other public and private variant databases.

This is achieved by what is referred to as variant normalization, a process that ensures the same variant is represented identically across different data sets. Performing variant normalization on terabytes of variant data in large projects using popular single-node tools can become quite a challenge as the acceptable input and output of these tools are the flat file formats that are commonly used to store variant calls (such as VCF and BGEN). To address this issue, we introduced the variant normalization transformation into Glow, which directly acts on a Spark Dataframe of variants to generate a DataFrame of normalized variants, harnessing the power of Spark to normalize variants from hundreds of thousands of samples in a fast and scalable manner with just a single line of Python or Scala code. Before addressing our normalizer, let us have a slightly more technical look at what variant normalization actually does.

6.4.1 What does variant normalization do?

Variant normalization ensures that the representation of a variant is both “parsimonious” and “left-aligned.” A variant is parsimonious if it is represented in as few nucleotides as possible without reducing the length of any allele to zero. An example is given in Fig. 6.17.

A variant is left-aligned if its position cannot be shifted to the left while keeping the length of all its alleles the same. An example is given in Fig. 6.18.

Tan et al. have proved that normalization results in uniqueness. In other words, two variants have different normalized representations if and only if they are actually different variants.
### Fig. 6.17: Variant parsimony

<table>
<thead>
<tr>
<th>Actual variation</th>
<th>Reference genome</th>
<th>Alternate allele</th>
<th>AAATCGCGTTA</th>
<th>AAGTGGCGTTA</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Some non-parsimonious presentations</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>POS 3</td>
<td>REF</td>
<td>ALT</td>
<td>ATCG</td>
<td>AGTG</td>
</tr>
<tr>
<td>POS 3</td>
<td>REF</td>
<td>ALT</td>
<td>ATC</td>
<td>AGT</td>
</tr>
<tr>
<td>POS 4</td>
<td>REF</td>
<td>ALT</td>
<td>TCG</td>
<td>GTG</td>
</tr>
<tr>
<td><strong>Parsimonious presentation</strong></td>
<td>POS 4</td>
<td>REF</td>
<td>TC</td>
<td>GT</td>
</tr>
<tr>
<td></td>
<td>ALT</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Fig. 6.18: Left-aligned variant

<table>
<thead>
<tr>
<th>Actual variation</th>
<th>Reference genome</th>
<th>Alternate allele</th>
<th>AAAGCGCGGT</th>
<th>AAGAGCGGTTT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Some non-left-aligned presentations</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>POS 7</td>
<td>REF</td>
<td>ALT</td>
<td>CGC</td>
<td>C</td>
</tr>
<tr>
<td>POS 5</td>
<td>REF</td>
<td>ALT</td>
<td>CGC</td>
<td>C</td>
</tr>
<tr>
<td>POS 4</td>
<td>REF</td>
<td>ALT</td>
<td>GCGGC</td>
<td>GCGC</td>
</tr>
<tr>
<td><strong>Normalized (left-aligned and parsimonious) presentation</strong></td>
<td>POS 3</td>
<td>REF</td>
<td>AGC</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>ALT</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Deletion of one GC
6.4.2 Variant normalization in Glow

We have introduced the `normalize_variants` transformer into Glow (Fig. 6.19). After ingesting variant calls into a Spark DataFrame using the VCF, BGEN or Delta readers, a user can call a single line of Python or Scala code to normalize all variants. This generates another DataFrame in which all variants are presented in their normalized form. The normalized DataFrame can then be used for downstream analyses like a GWAS using our built-in regression functions or an efficiently-parallelized GWAS tool.

The `normalize_variants` transformer brings unprecedented scalability and simplicity to this important upstream process, hence is yet another reason why Glow and Databricks UAP for Genomics are ideal platforms for biobank-scale genomic analyses, e.g., association studies between genetic variations and diseases across cohorts of hundreds of thousands of individuals.

6.4.3 The underlying normalization algorithm and its accuracy

There are several single-node tools for variant normalization that use different normalization algorithms. Widely used tools for variant normalization include `vt normalize`, `bcftools norm`, and the GATK’s `LeftAlignAndTrimVariants`.

Based on our own investigation and also as indicated by Bayat et al. and Tan et al., the GATK’s `LeftAlignAndTrimVariants` algorithm frequently fails to completely left-align some variants. For example, we noticed that on the `test_left_align_hg38.vcf` test file from GATK itself, applying `LeftAlignAndTrimVariants` results in an incorrect normalization of 3 of the 16 variants in the file, including the variants at positions `chr20:63669973`, `chr20:64012187`, and `chr21:13255301`. These variants are normalized correctly using `vt normalize` and `bcftools norm`.

Consequently, in our `normalize_variants` transformer, we used an improved version of the `bcftools norm` or `vt normalize` algorithms, which are similar in fundamentals. For a given variant, we start by right-trimming all the alleles of the variant as long as their rightmost nucleotides are the same. If the length of any allele reaches zero, we left-append it with a fixed block of nucleotides from the reference genome (the nucleotides are added in blocks as opposed to one-by-one to limit the number of referrals to the reference genome). When right-trimming is terminated, a potential left-trimming is performed to eliminate the leftmost nucleotides common to all alleles (possibly generated by prior left-appendings). The start, end, and alleles of the variants are updated appropriately during this process.

We benchmarked the accuracy of our normalization algorithm against `vt normalize` and `bcftools norm` on multiple test files and validated that our results match the results of these tools.

6.4. Streamlining Variant Normalization on Large Genomic Datasets
6.4.4 Optional splitting

Our `normalize_variants` transformer can optionally split multiallelic variants to biallelics. This is controlled by the mode option that can be supplied to this transformer. The possible values for the mode option are as follows:

- **normalize** (default), which performs normalization only,
- **split_and_normalize**, which splits multiallelic variants to biallelic ones before performing normalization, and
- **split**, which only splits multiallelics without doing any normalization.

The splitting logic of our transformer is the same as the splitting logic followed by GATK’s `LeftAlignAndTrimVariants` tool using the `--splitMultiallelics` option. More precisely, in case of splitting multiallelic variants loaded from VCF files, this transformer recalculates the GT blocks for the resulting biallelic variants if possible, and drops all INFO fields, except for `AC`, `AN`, and `AF`. These three fields are imputed based on the newly calculated GT blocks, if any exists, otherwise, these fields are dropped as well.

6.4.5 Using the transformer

Here, we briefly demonstrate how using Glow very large variant call sets can be normalized and/or split. First, VCF and/or BGEN files can be read into a Spark DataFrame as demonstrated in a prior post. This is shown in Python for the set of VCF files contained in a folder named `/databricks-datasets/genomics/call-sets`:

```python
original_variants_df = spark.read
  .format("vcf")
  .option("includeSampleIds", False)
  .load("/databricks-datasets/genomics/call-sets")
```

An example of the DataFrame `original_variants_df` is shown in Fig. 6.20.

![Fig. 6.20: The variant DataFrame original_variants_df](image)

The variants can then be normalized using the `normalize_variants` transformer as follows:

```python
import glow

ref_genome_path = "/mnt/dbnucleus/dbgenomics/grch38/data/GRCh38.fa"

normalized_variants_df = glow.transform(
    "normalize_variants",
    original_variants_df,
)
```

(continues on next page)
Note that normalization requires the reference genome .fasta or .fa file, which is provided using the reference_genome_path option. The .dict and .fai files must accompany the reference genome file in the same folder (read more about these file formats here).

Our example Dataframe after normalization can be seen in Fig. 6.21.

```
reference_genome_path=ref_genome_path
```

By default, the transformer normalizes each variant without splitting the multiallelic variants before normalization as seen in Fig. 6.21. By setting the mode option to split_and_normalize, nothing changes for biallelic variants, but the multiallelic variants are first split to the appropriate number of biallelics and the resulting biallelics are normalized. This can be done as follows:

```
split_and_normalized_variants_df = glow.transform(
    "normalize_variants",
    original_variants_df,
    reference_genome_path=ref_genome_path,
    mode="split_and_normalize"
)
```

The resulting DataFrame looks like Fig. 6.22.

As mentioned before, the transformer can also be used only for splitting of multiallelics without doing any normalization by setting the mode option to split.

### 6.4.6 Summary

Using Glow `normalize_variants` transformer, computational biologists and bioinformaticians can normalize very large variant datasets of hundreds of thousands of samples in a fast and scalable manner. Differently sourced call sets can be ingested and merged using VCF and/or BGEN readers, normalization can be performed using this transformer in a just a single line of code. The transformer can optionally perform splitting of multiallelic variants to biallelics as well.
Fig. 6.22: The `split_and_normalized_variants_df` DataFrame after applying normalize_variants transformer with `mode=split_and_normalize` on `original_variants_df`. Notice that for example the triallelic variant `chr20,start=19883344,end=19883345,REF=T,ALT=[TT,C]` of `original_variants_df` has been split into two biallelic variants and then normalized resulting in two normalized biallelic variants `chr20,start=19883336,end=19883337,REF=C,ALT=CT` and `chr20,start=19883344,end=19883345,REF=T,ALT=C`.

### 6.4.7 Try it!

Our `normalize_variants` transformer makes it easy to normalize (and split) large variant datasets with a very small amount of code. Learn more about other features of Glow here.

### 6.4.8 References

- Arash Bayat, Bruno Gaëta, Aleksandar Ignjatovic, Sri Parameswaran, Improved VCF normalization for accurate VCF comparison, Bioinformatics, Volume 33, Issue 7, 2017, Pages 964–970
CHAPTER SEVEN

ADDITIONAL RESOURCES

7.1 Databricks notebooks

Most of the code in the Databricks notebooks can be run on Spark and Glow alone, but some functions are only available on Databricks.

7.1.1 New to Databricks? Try Glow on Databricks for Free!

The Databricks Community Edition is free of charge. Follow our instructions to set up a Databricks Community Edition workspace and try the Glow documentation notebooks.

7.2 External blog posts

- Scaling Genomic Workflows with Spark SQL BGEN and VCF Readers
- Parallelizing SAIGE Across Hundreds of Cores
  - Parallelize SAIGE using Glow and the Pipe Transformer
- Accurately Building Genomic Cohorts at Scale with Delta Lake and Spark SQL
  - Joint genotyping with Glow and Databricks
- Introducing Glow: an open-source toolkit for large-scale genomic analysis
Glow’s Python API is designed to work seamlessly with PySpark and other tools in the Spark ecosystem. The functions here work with normal PySpark DataFrames and columns. You can access functions in any module from the top-level `glow` import.

### 8.1 Glow Top-Level Functions

**glow.glow.register** *(session: pyspark.sql.session.SparkSession)*

Register SQL extensions and py4j converters for a Spark session.

**Parameters**

- `session` – Spark session

**Example**

```python
>>> import glow

>>> glow.register(spark)
```

**glow.glow.transform** *(operation: str, df: pyspark.sql.dataframe.DataFrame, arg_map: Dict[str, Any] = None, **kwargs) → pyspark.sql.dataframe.DataFrame*

Apply a named transformation to a DataFrame of genomic data. All parameters apart from the input data and its schema are provided through the case-insensitive options map.

There are no bounds on what a transformer may do. For instance, it’s legal for a transformer to materialize the input DataFrame.

**Parameters**

- `operation` – Name of the operation to perform
- `df` – The input DataFrame
- `arg_map` – A string -> any map of arguments
- `kwargs` – Named arguments. If the arg_map is not specified, transformer args will be pulled from these keyword args.

**Example**

```python
>>> df = spark.read.format('vcf').load('test-data/1kg_sample.vcf')

>>> piped_df = glow.transform('pipe', df, cmd=['cat'], input_formatter='vcf', output_formatter='vcf', in_vcf_header='infer')
```
Returns The transformed DataFrame

8.2 Glow PySpark Functions

Glow includes a number of functions that operate on PySpark columns. These functions are interoperable with functions provided by PySpark or other libraries.

glow.functions.add_struct_fields(struct: Union[pyspark.sql.column.Column, str], *fields) → pyspark.sql.column.Column

Adds fields to a struct.

Added in version 0.3.0.

Examples

```py
>>> df = spark.createDataFrame([Row(struct=Row(a=1))])
>>> df.select(glow.add_struct_fields('struct', lit('b'), lit(2)).alias('struct')).collect()
[Row(struct=Row(a=1, b=2))]
```

Parameters
- **struct** – The struct to which fields will be added
- **fields** – The new fields to add. The arguments must alternate between string-typed literal field names and field values.

Returns A struct consisting of the input struct and the added fields

---

glow.functions.array_summary_stats(arr: Union[pyspark.sql.column.Column, str]) → pyspark.sql.column.Column

Computes the minimum, maximum, mean, standard deviation for an array of numerics.

Added in version 0.3.0.

Examples

```py
>>> df = spark.createDataFrame([Row(arr=[1, 2, 3])])
>>> df.select(glow.array_summary_stats('arr')).collect()
[Row(mean=2.0, stdDev=1.0, min=1.0, max=3.0)]
```

Parameters **arr** – An array of any numeric type

Returns A struct containing double mean, stdDev, min, and max fields

---

glow.functions.array_to_dense_vector(arr: Union[pyspark.sql.column.Column, str]) → pyspark.sql.column.Column

Converts an array of numerics into a spark.ml.DenseVector.

Added in version 0.3.0.
Examples

```python
>>> from pyspark.ml.linalg import DenseVector
>>> df = spark.createDataFrame([Row(arr=[1, 2, 3])])
>>> df.select(glow.array_to_dense_vector('arr').alias('v')).collect()
[Row(v=DenseVector([1.0, 2.0, 3.0]))]
```

Parameters `arr` – The array of numerics
Returns A `spark.ml.DenseVector`

glow.functions.array_to_sparse_vector (`arr` Union[pyspark.sql.column.Column, str]) \(\rightarrow\) pyspark.sql.column.Column
Converts an array of numerics into a `spark.ml.SparseVector`.
Added in version 0.3.0.

Examples

```python
>>> from pyspark.ml.linalg import SparseVector
>>> df = spark.createDataFrame([Row(arr=[1, 0, 2, 0, 3, 0])])
>>> df.select(glow.array_to_sparse_vector('arr').alias('v')).collect()
[Row(v=SparseVector(6, {0: 1.0, 2: 2.0, 4: 3.0}))]
```

Parameters `arr` – The array of numerics
Returns A `spark.ml.SparseVector`

glow.functions.call_summary_stats (`genotypes` Union[pyspark.sql.column.Column, str]) \(\rightarrow\) pyspark.sql.column.Column
Computes call summary statistics for an array of genotype structs. See `Variant Quality Control` for more details.
Added in version 0.3.0.

Examples

```python
>>> schema = 'genotypes: array<struct<calls: array<int>>>'
>>> df = spark.createDataFrame([Row(genotypes=[Row(calls=[0, 0]), Row(calls=[1, 1]), Row(calls=[1, 0])])], schema)
>>> df.select(glow.expand_struct(glow.call_summary_stats('genotypes'))).collect()
[Row(callRate=1.0, nCalled=3, nUncalled=0, nHet=1, nHomozygous=[1, 1], nNonRef=2, nAllelesCalled=6, alleleCounts=[3, 3], alleleFrequencies=[0.5, 0.5])]
```

Parameters `genotypes` – The array of genotype structs with `calls` field
Returns A struct containing `callRate`, `nCalled`, `nUncalled`, `nHet`, `nHomozygous`, `nNonRef`, `nAllelesCalled`, `alleleCounts`, `alleleFrequencies` fields. See `Variant Quality Control`.

glow.functions.dp_summary_stats (`genotypes` Union[pyspark.sql.column.Column, str]) \(\rightarrow\) pyspark.sql.column.Column
Computes summary statistics for the depth field from an array of genotype structs. See `Variant Quality Control`.
Added in version 0.3.0.
Examples

```python
>>> df = spark.createDataFrame([[Row(genotypes=[Row(depth=1), Row(depth=2), Row(depth=3)])], 'genotypes: array<struct<depth: int>>')
>>> df.select(glow.expand_struct(glow.dp_summary_stats('genotypes'))).collect()
[Row(mean=2.0, stdDev=1.0, min=1.0, max=3.0)]
```

Parameters `genotypes` – An array of genotype structs with `depth` field

Returns A struct containing `mean`, `stdDev`, `min`, and `max` of genotype depths

glow.functions.expand_struct(struct: Union[pyspark.sql.column.Column, str]) → pyspark.sql.column.Column
Promotes fields of a nested struct to top-level columns similar to using `struct.*` from SQL, but can be used in more contexts.
Added in version 0.3.0.

Examples

```python
>>> df = spark.createDataFrame([Row(struct=Row(a=1, b=2))])
>>> df.select(glow.expand_struct(col('struct'))).collect()
[Row(a=1, b=2)]
```

Parameters `struct` – The struct to expand

Returns Columns corresponding to fields of the input struct

glow.functions.explode_matrix(matrix: Union[pyspark.sql.column.Column, str]) → pyspark.sql.column.Column
Explodes a `spark.ml.Matrix` (sparse or dense) into multiple arrays, one per row of the matrix.
Added in version 0.3.0.

Examples

```python
>>> from pyspark.ml.linalg import DenseMatrix
>>> m = DenseMatrix(numRows=3, numCols=2, values=[1, 2, 3, 4, 5, 6])
>>> df = spark.createDataFrame([[Row(matrix=m)]]
>>> df.select(glow.explode_matrix('matrix').alias('row')).collect()
[Row(row=[1.0, 4.0]), Row(row=[2.0, 5.0]), Row(row=[3.0, 6.0])]
```

Parameters `matrix` – The `spark.ml.Matrix` to explode

Returns An array column in which each row is a row of the input matrix

glow.functions.genotype_states(genotypes: Union[pyspark.sql.column.Column, str]) → pyspark.sql.column.Column
Gets the number of alternate alleles for an array of genotype structs. Returns -1 if there are any -1 s (no-calls) in the calls array.
Added in version 0.3.0.
Examples

```python
>>> genotypes = [
    ... Row(calls=[1, 1]),
    ... Row(calls=[1, 0]),
    ... Row(calls=[0, 0]),
    ... Row(calls=[-1, -1])
]
>>> df = spark.createDataFrame([Row(genotypes=genotypes)], 'genotypes: array<struct<calls: array<int>>>')
>>> df.select(glow.genotype_states('genotypes').alias('states')).collect()
[Row(states=[2, 1, 0, -1])]
```

**Parameters**

- **genotypes** – An array of genotype structs with `calls` field

**Returns**

An array of integers containing the number of alternate alleles in each call array

---

**glow.functions.gq_summary_stats**

- **Parameters**
  - **genotypes** – The array of genotype structs with `conditionalQuality` field

- **Returns**
  - A struct containing `mean`, `stdDev`, `min`, and `max` of genotype qualities

---

**glow.functions.hard_calls**

- **Parameters**
  - **probabilities**
  - **numAlts**
  - **phased**

- **Returns**
  - A column containing hard calls

---

(continues on next page)
glow Documentation

(continued from previous page)

```python
>>> df.select(glow.hard_calls('probs', numAlts=lit(1), phased=lit(False)).alias('calls')).collect()
[Row(calls=[1, 0])]

>>> # Use the threshold parameter to change the minimum probability required for a call
>>> df = spark.createDataFrame([Row(probs=[0.05, 0.95, 0.0])])
>>> df.select(glow.hard_calls('probs', numAlts=lit(1), phased=lit(False), threshold=0.99).alias('calls')).collect()
[Row(calls=[-1, -1])]
```

Parameters

- **probabilities** – The array of probabilities to convert
- **numAlts** – The number of alternate alleles
- **phased** – Whether the probabilities are phased. If phased, we expect one $2 \times \text{numAlts}$ values in the probabilities array. If unphased, we expect one probability per possible genotype.
- **threshold** – The minimum probability to make a call. If no probability falls into the range of $[0, 1 - \text{threshold}]$ or $[\text{threshold}, 1]$, a no-call (represented by $-1$) will be emitted. If not provided, this parameter defaults to $0.9$.

Returns
An array of hard calls

```python
glow.functions.hardy_weinberg(genotypes: Union[pyspark.sql.column.Column, str]) -> pyspark.sql.column.Column
```
Computes statistics relating to the Hardy Weinberg equilibrium. See Variant Quality Control for more details.
Added in version 0.3.0.

**Examples**

```python
>>> genotypes = [
... Row(calls=[1, 1]),
... Row(calls=[1, 0]),
... Row(calls=[0, 0])
]
>>> df = spark.createDataFrame([Row(genotypes=genotypes)], 'genotypes: array<struct<calls: array<int>>>')
>>> df.select(glow.expand_struct(glow.hardy_weinberg('genotypes'))).collect()
[Row(hetFreqHwe=0.6, pValueHwe=0.7)]
```

Parameters **genotypes** – The array of genotype structs with **calls** field

Returns A struct containing two fields, **hetFreqHwe** (the expected heterozygous frequency according to Hardy-Weinberg equilibrium) and **pValueHwe** (the associated p-value)

```python
glow.functions.lift_over_coordinates(contigName: Union[pyspark.sql.column.Column, str], start: Union[pyspark.sql.column.Column, str], end: Union[pyspark.sql.column.Column, str], chainFile: str, minMatchRatio: float = None) -> pyspark.sql.column.Column
```
Performs liftover for the coordinates of a variant. To perform liftover of alleles and add additional metadata, see Liftover.
Added in version 0.3.0.
Examples

```python
>>> df = spark.read.format('vcf').load('test-data/liftover/unlifted.test.vcf').
  where('start = 18210071')
>>> chain_file = 'test-data/liftover/hg38ToHg19.over.chain.gz'
>>> reference_file = 'test-data/liftover/hg19.chr20.fa.gz'
>>> df.select('contigName', 'start', 'end').head()
Row(contigName='chr20', start=18210071, end=18210072)
>>> lifted_df = df.select(glow.expand_struct(glow.lift_over_coordinates(
  'contigName', 'start', 'end', chain_file)))
>>> lifted_df.head()
Row(contigName='chr20', start=18190715, end=18190716)
```

Parameters

- **contigName** – The current contig name
- **start** – The current start
- **end** – The current end
- **chainFile** – Location of the chain file on each node in the cluster
- **minMatchRatio** – Minimum fraction of bases that must remap to do liftover successfully. If not provided, defaults to 0.95.

Returns A struct containing contigName, start, and end fields after liftover

glow.functions.linear_regression_gwas (genotypes: Union[pyspark.sql.column.Column, str],
  phenotypes: Union[pyspark.sql.column.Column, str],
  covariates: Union[pyspark.sql.column.Column, str])
  → pyspark.sql.column.Column

Performs a linear regression association test optimized for performance in a GWAS setting. See Linear regression for details.

Added in version 0.3.0.

Examples

```python
>>> from pyspark.ml.linalg import DenseMatrix

>>> phenotypes = [2, 3, 4]
>>> genotypes = [0, 1, 2]
>>> covariates = DenseMatrix(numRows=3, numCols=1, values=[1, 1, 1])
>>> df = spark.createDataFrame([Row(genotypes=genotypes, phenotypes=phenotypes,
  covariates=covariates)])

>>> df.select(glow.expand_struct(glow.linear_regression_gwas('genotypes',
  'phenotypes', 'covariates'))).collect()
[Row(beta=0.9999999999999998, standardError=1.4901161193847656e-08, pValue=9.
  4863734847239922e-09)]
```

Parameters

- **genotypes** – A numeric array of genotypes
- **phenotypes** – A numeric array of phenotypes
- **covariates** – A spark.ml Matrix of covariates

Returns A struct containing beta, standardError, and pValue fields. See Linear regression.

Performs a logistic regression association test optimized for performance in a GWAS setting. See Logistic regression for more details.

Added in version 0.3.0.

Examples

```python
def from pyspark.ml.linalg import DenseMatrix
phenotypes = [1, 0, 0, 1, 1]
genotypes = [0, 0, 1, 2, 2]
covariates = DenseMatrix(numRows=5, numCols=1, values=[1, 1, 1, 1, 1])
df = spark.createDataFrame([Row(genotypes=genotypes, phenotypes=phenotypes, covariates=covariates)])
df.select(glow.expand_struct(glow.logistic_regression_gwas('genotypes', 'phenotypes', 'covariates', 'Firth'))).collect()
[Row(beta=0.7418937644793101, oddsRatio=2.09990848346903, waldConfidenceInterval=[0.2509874689201784, 17.569066925598555], pValue=0.3952193664732943)]
```

Parameters

- **genotypes** – An numeric array of genotypes
- **phenotypes** – A double array of phenotype values
- **covariates** – A spark.ml Matrix of covariates
- **test** – Which logistic regression test to use. Can be LRT or Firth

Returns A struct containing beta, oddsRatio, waldConfidenceInterval, and pValue fields. See Logistic regression.

---


Substitutes the missing values of a numeric array using the mean of the non-missing values. Any values that are NaN, null or equal to the missing value parameter are considered missing. See Variant data transformations for more details.

Added in version 0.4.0.

Examples

```python
def df = spark.createDataFrame([Row(unsubstituted_values=[float('nan'), None, 0.0, 1.0, 2.0, 3.0, 4.0])])
```

(continues on next page)
>>> df = spark.createDataFrame([Row(unsubstituted_values=[0, 1, 2, 3, -1, None])])
>>> df.select(glow.mean_substitute('unsubstituted_values').alias('substituted_values')).collect()
[Row(substituted_values=[0.0, 1.0, 2.0, 3.0, 1.5, 1.5])]

Parameters

- **array** – A numeric array that may contain missing values
- **missingValue** – A value that should be considered missing. If not provided, this parameter defaults to -1.

Returns A numeric array with substituted missing values

glow.functions.normalize_variant (contigName: Union[pyspark.sql.column.Column, str],

Normalizes the variant with a behavior similar to vt normalize or bcftools norm. Creates a StructType column including the normalized start, end, referenceAllele and alternateAlleles fields (whether they are changed or unchanged as the result of normalization) as well as a StructType field called normalizationStatus that contains the following fields:

- **changed** - A boolean field indicating whether the variant data was changed as a result of normalization
- **errorMessage** - An error message in case the attempt at normalizing the row hit an error. In this case, the changed field will be set to False. If no errors occur, this field will be null.

In case of an error, the start, end, referenceAllele and alternateAlleles fields in the generated struct will be null.

Added in version 0.3.0.

Examples

```python
>>> df = spark.read.format('vcf').load('test-data/variantsplitternormalizer-test/test_left_align_hg38_altered.vcf')
>>> ref_genome = 'test-data/variantsplitternormalizer-test/Homo_sapiens_assembly38.20.21_altered.fasta'
>>> df.select('contigName', 'start', 'end', 'referenceAllele', 'alternateAlleles').head()
Row(contigName='chr20', start=400, end=401, referenceAllele='G', alternateAlleles=['GATCTCCTCCTTCTTCTAATACAAACACATGACTCTGTGGTCTCTTAGTTACTGTTGAG'])
>>> normalized_df = df.select('contigName', glow.expand_struct(glow.normalize_variant('contigName', 'start', 'end', 'referenceAllele', 'alternateAlleles', ref_genome))).head()
Row(contigName='chr20', start=268, end=269, referenceAllele='A', alternateAlleles=['ATTTGAGATCTCCTCCTTCTTCTTCTAATACAAACACATGACTCTGTGGTCTCTTAGTTACTGTTGAG'], normalizationStatus=Row(changed=True, errorMessage=None))
```
Parameters

- **contigName** – The current contig name
- **start** – The current start
- **end** – The current end
- **refAllele** – The current reference allele
- **altAlleles** – The current array of alternate alleles
- **refGenomePathString** – A path to the reference genome .fasta file. The .fasta file must be accompanied with a .fai index file in the same folder.

Returns A struct as explained above

```python
```

Computes per-sample call summary statistics. See Sample Quality Control for more details.

Added in version 0.3.0.

Examples

```python
>>> sites = ...
  ... Row(refAllele='C', alternateAlleles=['G'], genotypes=[Row(sampleId='NA12878',
  ... calls=[0, 0])]),
  ...
  ... Row(refAllele='A', alternateAlleles=['G'], genotypes=[Row(sampleId='NA12878',
  ... calls=[1, 1])]),
  ...
  ... Row(refAllele='AG', alternateAlleles=['A'], genotypes=[Row(sampleId='NA12878',
  ...
  ... calls=[1, 0])])
>>> df = spark.createDataFrame(sites, 'alternateAlleles: array<string>,
  ...
  ... genotypes: array<struct<sampleId: string, calls: array<int>>>), refAllele: string
  ...
  ... ->')
>>> df.select(glow.sample_call_summary_stats('genotypes', 'refAllele',
  ...
  ... 'alternateAlleles').alias('stats')).collect()
[Row(stats=Row(sampleId='NA12878', callRate=1.0, nCalled=3, nUncalled=0,
  ...
  ... nHomRef=1, nHet=1, nHomVar=1, nSnp=2, nInsertion=0, nDeletion=1, nTransition=2,
  ...
  ... nTransversion=0, nSpanningDeletion=0, rTiTv=inf, rInsertionDeletion=0.0, rHetHomVar=1.0))]
```
glow.functions.sample_dp_summary_stats (genotypes: \[\text{pyspark.sql.column.Column, str}\]) \rightarrow \text{pyspark.sql.column.Column}

Computes per-sample summary statistics about the depth field in an array of genotype structs.

Added in version 0.3.0.

**Examples**

```python
>>> sites = [
    ... Row(genotypes=[Row(sampleId='NA12878', depth=1)]),
    ... Row(genotypes=[Row(sampleId='NA12878', depth=2)]),
    ... Row(genotypes=[Row(sampleId='NA12878', depth=3)])
    
>>> df = spark.createDataFrame(sites, 'genotypes: array<struct<depth: int, sampleId: string>>')
>>> df.select(glow.sample_dp_summary_stats('genotypes').alias('stats')).collect()
[Row(stats=[Row(sampleId='NA12878', mean=2.0, stdDev=1.0, min=1.0, max=3.0)])]
```

**Parameters**

- **genotypes** – An array of genotype structs with depth field

**Returns**

An array of structs where each struct contains mean, stDev, min, and max of the genotype depths for a sample. If sampleId is present in a genotype, it will be propagated to the resulting struct as an extra field.

---

glow.functions.sample_gq_summary_stats (genotypes: \[\text{pyspark.sql.column.Column, str}\]) \rightarrow \text{pyspark.sql.column.Column}

Computes per-sample summary statistics about the genotype quality field in an array of genotype structs.

Added in version 0.3.0.

**Examples**

```python
>>> sites = [
    ... Row(genotypes=[Row(sampleId='NA12878', conditionalQuality=1)]),
    ... Row(genotypes=[Row(sampleId='NA12878', conditionalQuality=2)]),
    ... Row(genotypes=[Row(sampleId='NA12878', conditionalQuality=3)])
    
>>> df = spark.createDataFrame(sites, 'genotypes: array<struct<conditionalQuality: int, sampleId: string>>')
>>> df.select(glow.sample_gq_summary_stats('genotypes').alias('stats')).collect()
[Row(stats=[Row(sampleId='NA12878', mean=2.0, stdDev=1.0, min=1.0, max=3.0)])]
```

**Parameters**

- **genotypes** – An array of genotype structs with conditionalQuality field

**Returns**

An array of structs where each struct contains mean, stDev, min, and max of the genotype qualities for a sample. If sampleId is present in a genotype, it will be propagated to the resulting struct as an extra field.

---

glow.functions.subset_struct (struct: \[\text{pyspark.sql.column.Column, str}\], *fields) \rightarrow \text{pyspark.sql.column.Column}

Selects fields from a struct.

Added in version 0.3.0.
Examples

```python
>>> df = spark.createDataFrame([Row(struct=Row(a=1, b=2, c=3))])
>>> df.select(glow.subset_struct('struct', 'a', 'c').alias('struct')).collect()
[Row(struct=Row(a=1, c=3))]
```

Parameters

- **struct** – Struct from which to select fields
- **fields** – Fields to select

Returns

A struct containing only the indicated fields

---

glow.functions.vector_to_array(vector: Union[pyspark.sql.column.Column, str]) → pyspark.sql.column.Column

Converts a spark.mlVector (sparse or dense) to an array of doubles.

Added in version 0.3.0.

Examples

```python
>>> from pyspark.ml.linalg import DenseVector, SparseVector
>>> df = spark.createDataFrame([Row(v=SparseVector(3, {0: 1.0, 2: 2.0})), Row(v=DenseVector([3.0, 4.0]))])
>>> df.select(glow.vector_to_array('v').alias('arr')).collect()
[Row(arr=[1.0, 0.0, 2.0]), Row(arr=[3.0, 4.0])]
```

Parameters

- **vector** – Vector to convert

Returns

An array of doubles

---

8.3 GloWGR

8.3.1 Data preparation and helper functions


Creates a blocked GT matrix and index mapping from sample blocks to a list of corresponding sample IDs. Uses the same sample-blocking logic as the blocked GT matrix transformer.

Requires that:
- Each variant row has the same number of values
- The number of values per row matches the number of sample IDs

Parameters

- **variant_df** – The variant DataFrame
- **sample_ids** – The list of sample ID strings
- **variants_per_block** – The number of variants per block
• **sample_block_count** – The number of sample blocks

**Returns** tuple of (blocked GT matrix, index mapping)

glow.wgr.functions.get_sample_ids(data: pyspark.sql.dataframe.DataFrame) \(\rightarrow\) List[str]

Extracts sample IDs from a variant DataFrame, such as one read from PLINK files.

**Requires that the sample IDs:**

• Are in genotype.sampleId

• Are the same across all the variant rows

• Are a list of strings

• Are non-empty

• Are unique

**Parameters**

- **data** – The variant DataFrame containing sample IDs

**Returns** list of sample ID strings


Reshapes a Pandas DataFrame into a Spark DataFrame with a convenient format for Glow’s GWAS functions. This function can handle labels that are either per-sample or per-sample and per-contig, like those generated by GloWGR’s transform_loco function.

**Examples**

```python
>>> label_df = pd.DataFrame({'label1': [1, 2], 'label2': [3, 4]}, index=['sample1', 'sample2'])
>>> reshaped = reshape_for_gwas(spark, label_df)
>>> reshaped.head()
Row(label='label1', values=[1, 2])
```

```python
>>> loco_label_df = pd.DataFrame({'label1': [1, 2], 'label2': [3, 4]},
...    index=pd.MultiIndex.from_tuples([('sample1', 'chr1'), ('sample1', 'chr2')]))
>>> reshaped = reshape_for_gwas(spark, loco_label_df)
>>> reshaped.head()
Row(label='label1', contigName='chr1', values=[1])
```

**Requires that:**

• The input label DataFrame is indexed by sample id or by (sample id, contig name)

**Parameters**

- **spark** – A Spark session

- **label_df** – A pandas DataFrame containing labels. The Data Frame should either be indexed by sample id or multi indexed by (sample id, contig name). Each column is interpreted as a label.

**Returns** A Spark DataFrame with a convenient format for Glow regression functions. Each row contains the label name, the contig name if provided in the input DataFrame, and an array containing the label value for each sample.
8.3.2 Ridge model

class glow.wgr.linear_model.ridge_model.RidgeReducer

The RidgeReducer class is intended to reduce the feature space of an $N \times M$ block matrix $X$ to an $N \times P < M$ block matrix. This is done by fitting $K$ ridge models within each block of $X$ on one or more target labels, such that a block with $L$ columns to begin with will be reduced to a block with $K$ columns, where each column is the prediction of one ridge model for one target label.


Fits a ridge reducer model, represented by a Spark DataFrame containing coefficients for each of the ridge alpha parameters, for each block in the starting matrix, for each label in the target labels.

**Parameters**

- **blockdf** – Spark DataFrame representing the beginning block matrix $X$
- **labelfd** – Pandas DataFrame containing the target labels used in fitting the ridge models
- **sample_blocks** – Dict containing a mapping of sample_block ID to a list of corresponding sample IDs
- **covdf** – Pandas DataFrame containing covariates to be included in every model in the stacking ensemble (optional).

**Returns** Spark DataFrame containing the model resulting from the fitting routine.


Fits a ridge reducer model with a block matrix, then transforms the matrix using the model.

**Parameters**

- **blockdf** – Spark DataFrame representing the beginning block matrix $X$
- **labelfd** – Pandas DataFrame containing the target labels used in fitting the ridge models
- **sample_blocks** – Dict containing a mapping of sample_block ID to a list of corresponding sample IDs
- **covdf** – Pandas DataFrame containing covariates to be included in every model in the stacking ensemble (optional).

**Returns** Spark DataFrame representing the reduced block matrix


Transforms a starting block matrix to the reduced block matrix, using a reducer model produced by the RidgeReducer fit method.

**Parameters**

- **blockdf** – Spark DataFrame representing the beginning block matrix
- **labelfd** – Pandas DataFrame containing the target labels used in fitting the ridge models
- **sample_blocks** – Dict containing a mapping of sample_block ID to a list of corresponding sample IDs
modeldf – Spark DataFrame produced by the RidgeReducer fit method, representing the reducer model

covdf – Pandas DataFrame containing covariates to be included in every model in the stacking ensemble (optional).

Returns Spark DataFrame representing the reduced block matrix

class glow.wgr.linear_model.ridge_model.RidgeRegression

The RidgeRegression class is used to fit ridge models against one or labels optimized over a provided list of ridge alpha parameters. It is similar in function to RidgeReducer except that whereas RidgeReducer attempts to reduce a starting matrix X to a block matrix of smaller dimension, RidgeRegression is intended to find an optimal model of the form $Y_{\text{hat}} \sim XB$, where $Y_{\text{hat}}$ is a matrix of one or more predicted labels and B is a matrix of coefficients. The optimal ridge alpha value is chosen for each label by maximizing the average out of fold r2 score.

Parameters

- blockdf – Spark DataFrame representing the beginning block matrix X
- labeldf – Pandas DataFrame containing the target labels used in fitting the ridge models
- sample_blocks – Dict containing a mapping of sample_block ID to a list of corresponding sample IDs
- covdf – Pandas DataFrame containing covariates to be included in every model in the stacking ensemble (optional).

Returns Two Spark DataFrames, one containing the model resulting from the fitting routine and one containing the results of the cross validation procedure.


Fits a ridge regression model with a block matrix, then transforms the matrix using the model.

Parameters

- blockdf – Spark DataFrame representing the beginning block matrix X
- labeldf – Pandas DataFrame containing the target labels used in fitting the ridge models
- sample_blocks – Dict containing a mapping of sample_block ID to a list of corresponding sample IDs
- covdf – Pandas DataFrame containing covariates to be included in every model in the stacking ensemble (optional).

Returns Pandas DataFrame containing prediction $y_{\text{hat}}$ values. The shape and order match labeldf such that the rows are indexed by sample ID and the columns by label. The column types are float64.
**transform**


Generates predictions for the target labels in the provided label DataFrame by applying the model resulting from the RidgeRegression fit method to the starting block matrix.

**Parameters**

- **blockdf** – Spark DataFrame representing the beginning block matrix X
- **labelfd** – Pandas DataFrame containing the target labels used in fitting the ridge models
- **sample_blocks** – Dict containing a mapping of sample_block ID to a list of corresponding sample IDs
- **modeldf** – Spark DataFrame produced by the RidgeRegression fit method, representing the reducer model
- **cvdf** – Spark DataFrame produced by the RidgeRegression fit method, containing the results of the cross
- **routine.** (validation) –
- **cvdfr** – Pandas DataFrame containing covariates to be included in every model in the stacking ensemble (optional).

**Returns** Pandas DataFrame containing prediction y_hat values. The shape and order match labelfd such that the rows are indexed by sample ID and the columns by label. The column types are float64.

**transform_loco**


Generates predictions for the target labels in the provided label DataFrame by applying the model resulting from the RidgeRegression fit method to the starting block matrix using a leave-one-chromosome-out (LOCO) approach.

**Parameters**

- **blockdf** – Spark DataFrame representing the beginning block matrix X
- **labelfd** – Pandas DataFrame containing the target labels used in fitting the ridge models
- **sample_blocks** – Dict containing a mapping of sample_block ID to a list of corresponding sample IDs
- **modeldf** – Spark DataFrame produced by the RidgeRegression fit method, representing the reducer model
- **cvdf** – Spark DataFrame produced by the RidgeRegression fit method, containing the results of the cross
- **routine.** (validation) –
- **cvdfr** – Pandas DataFrame containing covariates to be included in every model in the stacking
- **ensemble** (optional) –
- **chromosomes** – List of chromosomes for which to generate a prediction (optional). If not provided, the
• will be inferred from the block matrix. (chromosomes)

**Returns** Pandas DataFrame containing prediction \( y \_\text{hat} \) values per chromosome. The rows are indexed by sample ID and chromosome; the columns are indexed by label. The column types are float64. The DataFrame is sorted using chromosome as the primary sort key, and sample ID as the secondary sort key.
g
  glow.functions, 72
  glow.glow, 71
  glow.wgr.functions, 82
  glow.wgr.linear_model.ridge_model, 84
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_struct_fields()</td>
<td>block_variants_and_samples()</td>
<td>call_summary_stats()</td>
<td>dp_summary_stats()</td>
<td>expand_struct()</td>
<td>fit()</td>
<td>genotype_states()</td>
</tr>
<tr>
<td>(in module glow.functions), 72</td>
<td>(in module glow.wgr.functions), 82</td>
<td>(in module glow.functions), 73</td>
<td>(in module glow.functions), 73</td>
<td>(in module glow.functions), 74</td>
<td>(glow.wgr.linear_model.ridge_model.RidgeReducer method), 84</td>
<td>(in module glow.functions), 74</td>
</tr>
<tr>
<td>array_summary_stats()</td>
<td></td>
<td></td>
<td></td>
<td>explode_matrix()</td>
<td>fit()</td>
<td>(glow.wgr.linear_model.ridge_model.RidgeRegression method), 85</td>
</tr>
<tr>
<td>(in module glow.functions), 72</td>
<td></td>
<td></td>
<td></td>
<td>(in module glow.wgr.functions), 83</td>
<td></td>
<td></td>
</tr>
<tr>
<td>array_to_dense_vector()</td>
<td></td>
<td></td>
<td></td>
<td>fit_transform()</td>
<td>fit_transform()</td>
<td>(glow.wgr.linear_model.ridge_model.RidgeRegression method), 85</td>
</tr>
<tr>
<td>(in module glow.functions), 72</td>
<td></td>
<td></td>
<td></td>
<td>(glow.wgr.linear_model.ridge_model.RidgeReducer method), 84</td>
<td></td>
<td></td>
</tr>
<tr>
<td>array_to_sparse_vector()</td>
<td></td>
<td></td>
<td></td>
<td>reshape_for_gwas()</td>
<td>register()</td>
<td>(in module glow.wgr.functions), 83</td>
</tr>
<tr>
<td>(in module glow.functions), 73</td>
<td></td>
<td></td>
<td></td>
<td>(in module glow.wgr.functions), 83</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>H</th>
<th>L</th>
<th>M</th>
<th>N</th>
<th>R</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>hard_calls()</td>
<td>lift_over_coordinates()</td>
<td>mean_substitute()</td>
<td>normalize_variant()</td>
<td>register()</td>
<td>sample_call_summary_stats()</td>
</tr>
<tr>
<td>(in module glow.functions), 75</td>
<td>(in module glow.functions), 76</td>
<td>(in module glow.functions), 78</td>
<td>(in module glow.functions), 79</td>
<td>(in module glow.glow), 71</td>
<td>(in module glow.wgr.functions), 80</td>
</tr>
<tr>
<td>hardy_weinberg()</td>
<td>linear_regression_gwas()</td>
<td>(in module glow.functions), 77</td>
<td>(in module glow.functions), 79</td>
<td>reshape_for_gwas()</td>
<td>sample_dp_summary_stats()</td>
</tr>
<tr>
<td>(in module glow.functions), 76</td>
<td>(in module glow.functions), 78</td>
<td></td>
<td></td>
<td>(in module glow.wgr.functions), 83</td>
<td>(in module glow.functions), 80</td>
</tr>
<tr>
<td>gq_summary_stats()</td>
<td>logistic_regression_gwas()</td>
<td></td>
<td></td>
<td>RidgeReducer</td>
<td>sample_gq_summary_stats()</td>
</tr>
<tr>
<td>(in module glow.functions), 75</td>
<td>(in module glow.functions), 78</td>
<td></td>
<td></td>
<td>(class in glow.wgr.linear_model.ridge_model), 84</td>
<td>(in module glow.functions), 81</td>
</tr>
<tr>
<td>ridge_regression_gwas()</td>
<td>(in module glow.functions), 77</td>
<td></td>
<td></td>
<td>RidgeRegression</td>
<td></td>
</tr>
<tr>
<td>(in module glow.functions), 77</td>
<td>(in module glow.functions), 78</td>
<td></td>
<td></td>
<td>(class in glow.wgr.linear_model.ridge_model), 85</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>G</th>
<th></th>
<th></th>
<th></th>
<th>R</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_sample_ids()</td>
<td></td>
<td></td>
<td></td>
<td>register()</td>
<td>sample_call_summary_stats()</td>
</tr>
<tr>
<td>(in module glow.wgr.functions), 83</td>
<td></td>
<td></td>
<td></td>
<td>(in module glow.wgr.functions), 80</td>
<td></td>
</tr>
<tr>
<td>glow.functions (module), 72</td>
<td></td>
<td></td>
<td></td>
<td>sample_dp_summary_stats()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(in module glow.wgr.functions), 80</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sample_gq_summary_stats()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(in module glow.wgr.functions), 81</td>
<td></td>
</tr>
</tbody>
</table>
subset_struct() (in module glow.functions), 81

T
transform() (glow.wgr.linear_model.ridge_model.RidgeReducer method), 84
transform() (glow.wgr.linear_model.ridge_model.RidgeRegression method), 85
transform() (in module glow.glow), 71
transform_loco() (glow.wgr.linear_model.ridge_model.RidgeRegression method), 86

V
vector_to_array() (in module glow.functions), 82